NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE IMPLEMENTATION OF A
MULTI-LINGUAL DATABASE SYSTEM--
MULTI-BACKEND DATABASE SYSTEM INTERFACE

by

Steven Todd Holste

June 1986

Thesis Advisor: David K. Hsiao

URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION VAL POSTGRADUATE SCHOOL 6b. OFFICE SYMBOL (If applicable) 52	7a. NAME OF MONITORING ORGANIZATION NAVAL POSTGRADUATE SCHOOL
ADDRESS (City, State, and ZIP Code) NTEREY, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) MONTEREY, CA 93943-5000
NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

TITLE (Include Security Classification) UNCLASSIFIED
E IMPLEMENTATION OF A MULTI-LINGUAL DATABASE SYSTEM--MULTI-BACKEND
TABASE SYSTEM INTERFACE

PERSONAL AUTHOR(S)
STEVEN TODD HOLSTE

TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
ster's Thesis	FROM _____ TO _____	1986 June 20	128

SUPPLEMENTARY NOTATION

COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Database Management System, Multi-Lingual Database System, Multi-Backend Database System, Relational Data Model, Hierarchical Data (Cont)

ABSTRACT (Continue on reverse if necessary and identify by block number)
e limitations of the traditional Database Management System (DBMS) have come increasingly clear in recent years. Some of these limitations are terface inflexibility for user accesses, mono-lingual restriction in ta languages, performance degradations over time, and excessive costs upgrading.
o complementary approaches to the DBMS design and implementation--the lti-lingual database system (MLDS) and the multi-backend database stem (MBDS)--effectively deal with the limitations of the traditional MS approach. MLDS offers a multi-lingual capability to the DBMS en-ronment, thus freeing the user from the limitations and inflexibility the single-data-model-and-language approach. MBDS, by contast, is signed to deal with the issues of performance degradation and upgrading sts by providing a parallel processing capability, and utilizing (Cont)

DISTRIBUTION / AVAILABILITY OF ABSTRACT X UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
NAME OF RESPONSIBLE INDIVIDUAL of. David K. Hsiao	22b. TELEPHONE (Include Area Code) 408-646-2168 22c. OFFICE SYMBOL 52Hq

FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

BLOCK 18 (Continued)

Model, Network Data Model, Template File, Descriptor File.

BLOCK 19 (Continued)

replicated software and identical hardware for expansion. System upgrades with MBDS have been shown to provide an essentially proportional performance gain-to-upgrade-cost ratio.
In this thesis, we present the implementation of an interface between MLDS and MBDS. Specifically, we present the procedures which create the Template and Descriptor Files in MLDS that are required by MBDS. Additionally, we describe the integration process tying these two systems together.

The Implementation of a Multi-Lingual Database System -- Multi-Backend Database System Interface

by

Steven T. Holste
Captain. United States Marine Corps
B.S., University of Washington, 1976
M.B.A., University of Washington, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Abstract

The limitations of the traditional Database Management System (DBMS) have become increasingly clear in recent years. Some of these limitations are interface inflexibility for user accesses, mono-lingual restriction in data languages, performance degradations over time, and excessive costs in upgrading.

Two complementary approaches to the DBMS design and implementation--the multi-lingual database system (MLDS) and the multi-backend database system (MBDS)--effectively deal with the limitations of the traditional DBMS approach. MLDS offers a multi-lingual capability to the DBMS environment, thus freeing the user from the limitations and inflexibility of the single-data-model-and-language approach. MBDS, by contrast, is designed to deal with the issues of performance degradation and upgrading costs by providing a parallel processing capability, and utilizing replicated software and identical hardware for expansion. System upgrades with MBDS have been shown to provide an essentially proportional performance gain-to-upgrade-cost ratio.

In this thesis, we present the implementation of an interface between MLDS and MBDS. Specifically, we present the procedures which create the Template and Descriptor Files in MLDS that are required by MBDS. Additionally, we describe the integration process tying these two systems together.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

Database Management Systems (DBMSs)--as traditionally designed, implemented, and utilized--typically share a number of common features. Generally, a certain DBMS package (which is written to conform to one of the popular, prevailing data models) is selected and installed into the organization's "mainframe" computer. Users' transactions are executed in the mainframe together with all of an installation's other processes: database files are stored in the generally-shared secondary-storage devices. As the inevitable database file growth and database applications increases occur, *all* users of the system begin to suffer noticeable performance degradation. This has ordinarily been viewed as the "price that one must pay," if one is to operate a DBMS.

Among the current DBMS data models are the hierarchical, the network, the relational, the entity-relationship, and the attribute-based data models. Some of these DBMSs include the Information Management System (IMS), an IBM product, supporting the hierarchical data model and its companion data manipulation language, Data Language/I (DL/I). Similarly, the network model is supported by Sperry Univac's DMS-1100, together with the network model-based data manipulation language, CODASYL Data Manipulation Language (CODASYL-DML). Another commercial product is IBM's SQL/Data System, supporting both the relational data model and the relational model-based data definition and manipulation language, Structured English Query Language (SQL).

Each of these models naturally tends to have its own specific strengths and weaknesses. The comparative ease of configuring a useful and representative database: the kinds of data--the objects, their properties, and the relationships between them--that may be conveniently and clearly represented: and the ease and clarity of performing the desired manipulations upon the database (e.g., retrievals, insertions, deletions, and updates) are examples of common metrics of the "fitness" of a DBMS for a specific application. While it is clear that no single data model-based DBMS will perform optimally in every application, this has frequently been the price that an organization with limited resources (both financial and computational) has had to pay in selecting a model. An organization may choose to satisfice, based on a view of which model best meets its data management requirements, on familiarity or experience with one or another of the models, on cost or availability of the competing DBMSs, on existing or projected computational and/or storage resource limitations, or on any combination of these--and other--criteria.

Unfortunately, it is rare that an organization can afford to institute two or more of the available model-based systems. Besides the obvious increases in direct costs that such a procurement reflects are the computational and storage costs. The computational costs involved include the competition of two or more computation-intensive DBMSs which may be concurrently attempting to execute: storage costs may be envisioned in terms of the explosive, exponential increase in storage requirements as the *same data*-- or portions thereof--are repetitively stored in secondary storage, according to the requirements of the various data models. The impact on primary storage, with concurrently executing transactions, is similarly predictable.

A final difficulty that we may discuss here involves performance upgrades. It may be observed that the standard von Neumann architecture has evolved into a machine generally capable of handling a wide variety of computational tasks. At the same time, the typical mainframe thus generalized will rarely be optimally suited to handle any given, specific task. This is certainly the case in terms of executing a DBMS. An organization's computer will ordinarily be involved in a variety of tasks, attempting to meet the needs of a diverse community of users. Experience has shown that DBMS operations typically load a system considerably, resulting in increasingly degraded performance (as measured in terms of the response time, turnaround, and throughput) for *all* users.

Faced with deteriorating performance--and often with simultaneously increasing computational demands, by both DBMS- and non-DBMS-users--the need to seek relief through some form of upgrade rapidly emerges as an urgent requirement. The traditional approach has often been to upgrade or replace the mainframe--an extremely expensive method, often yielding only incremental improvement. Presumably, the cost of this expansion (which may be required solely to offset increasing DBMS use) will be shared throughout the organization at large, potentially creating an equity issue.

B. ALTERNATIVE APPROACHES

In this thesis, we are concerned with implementation issues pertaining to alternative approaches to the foregoing difficulties. Specifically, we introduce and discuss a multi-lingual database system (MLDS) and a multi-backend database system (MBDS) that, together, represent a significant advance over traditional and existing DBMS methodologies.

In the MLDS, a single data model (the attribute-based data model, originally described by Hsiao in [Ref. 1] and extended by Wong [Ref. 2]) is actually implemented in the computer system. As discussed subsequently, it is in fact the attribute-based data model that MBDS utilizes in storing a database, and in performing transactions against that database. In operation, it is possible to create and store databases, and subsequently to manipulate them, utilizing not just the attribute-based data model, but the relational, hierarchical, network, and entity-relationship models as well. The attribute-based model is thus both conceptually simple *and* exceedingly powerful, as demonstrated by its capability of effectively realizing this diverse collection of data models in its software.

As will be discussed in more detail in succeeding sections, MLDS achieves its goal of supporting the various models and languages *not* by the proliferation of DBMSs and multiply-stored databases alluded to previously, but rather by storing a given database once, and providing an interface that various users may utilize to access this database. Using this interface (the Language Interface Layer, or LIL), a database may be initially *created*, *indexed*, and *stored* according to any of the supported models--and subsequently *manipulated* by any of these model-based languages as well. The power and flexibility of MLDS are thus apparent.

MBDS, on the other hand, attacks both the performance problems and the upgrade difficulties that inhere in the traditional approach to both database and installation management. As is also described in more detail subsequently, the MBDS approach is to download the vast majority of DBMS operations *from* the mainframe *to* a series of identical "backend" computers. These backends may increase performance at a much smaller cost than the cost which may be realized through the traditional mainframe enhancement/replacement approaches.

Despite this downloading, the user will continue to interact through the mainframe (i.e., the host) as before, using the same terminals and interacting in essentially the same way. Thus, a high degree of transparency is achieved as users execute DBMS functions utilizing the model(s) of their choice, dealing with the familiar operating system of the host mainframe; meanwhile, MLDS and MBDS are operating together to permit this multi-lingual capability, and to dramatically enhance performance characteristics for DBMS- and non-DBMS-users alike.

C. THESIS OVERVIEW

The purpose of this thesis is to implement an interface between MLDS and MBDS, in a series of programs written in the C programming language. This interface consists of a pair of files: the Template File and the Descriptor File, written for each of the three supported models--the relational, hierarchical, and network models.

In Chapter II, MLDS and MBDS are discussed in some detail. Here, we learn about the native, or *kernel*, model and language of MBDS, i.e. the attribute-based data model and language. In Chapter III, we introduce and discuss the data-model transformations that permit a user to select a data model of his choice, via MLDS.

Next, Chapter IV defines the algorithms, together with the specific implementation requirements (i.e., C programs), that form the MLDS-MBDS interface. Some of the software engineering issues of this integration are then discussed in Chapter V, while Chapter VI presents conclusions drawn from the research and implementation of this thesis.

Appendices A through F are the various programs that implement the MLDS-MBDS interface through the creation of the Template and the Descriptor Files.

II. A DESCRIPTION OF MLDS AND MBDS

In this chapter, we discuss some of the rationale underlying the multi-lingual database system (MLDS)--the design goals, the comparative advantages it offers, and some of the additional operational functionalities that it provides. With this background, we then briefly review the MLDS structure, and observe the means by which it carries out its tasks.

Concerning the multi-backend database system (MBDS), the motivation and background are reviewed. We then discuss its hardware configuration and operations.

A. THE MULTI-LINGUAL DATABASE SYSTEM (MLDS)

1. Motivation and Goals

Historically, the approach taken in the selection and utilization of a database management system (DBMS) has followed one of two rather general paths:

1. Selection of a specific data model, followed by the selection of a corresponding model-based data manipulation language; or

2. Selection of a preferred data manipulation language system. At this point, all concerns pertaining to which is an appropriate or desirable data model become moot--or rather, become non-issues, prescribed by default.

In either case, the selected system (for example, IBM's relational-model based SQL/DS) is then installed on the computer, corresponding model-based databases

are developed and stored, and corresponding model-based transactions and queries are written and run against the respective databases.

This methodology would be perfectly acceptable (in fact, desirable) to the extent that:

1. A given (single) DBMS is ideally suited to model all of an organization's data, the properties of the data, and the interrelationships between them;

2. All users and potential users of the system are qualified, skilled, and comfortable with the chosen system; and

3. The present, ideal circumstances will *never* change.

Questions of intra- and inter-organizational compatibility, together with those of economics, are highly individual in nature, and therefore will not be addressed here.

In reality, however, it is extremely unlikely that *any* (let alone all) of these conditions will ever prevail. The typical organization, for example, will quite likely have data that reflects an hierarchical structure, together with data most appropriately modeled by the relational data model, and so forth. In the case of the second condition, it is reasonable to expect (within any normally diverse population of users) varying degrees of familiarity and expertise with the differing DBMSs. Even if the first two conditions can be reasonably met, the third condition (permanent stability) is unlikely to be met, since databases *do* grow, and applications *do* change over time.

Is this mono-lingual approach, in fact, desirable? An interesting analogy is offered by Demurjian and Hsiao [Ref. 3: pp. 2-3] with respect to Operating Systems. As with most current DBMSs, early Operating Systems (such as the Fortran Monitor System of the 1950's) were essentially mono-lingual, supporting but a single programming language (such as FORTRAN). As operating systems

have evolved; this insistence on a single language has given way to operating systems capable of supporting a large number and variety of programming languages.

Thus, some of the principal tasks of an operating system might be characterized as multi-lingual in providing multiple programming languages and operational modes (such as interactive and batch processing), as well as resource allocation. Likewise, the modern DBMS should be able to recognize and execute transactions for different data models in their corresponding data manipulation languages; offer such modes of database access as ad-hoc queries and transaction processing; and carefully oversee the access to, and management of, its principal resource, i.e., its databases. Logically pursuing this analogy, then, it seems clear that the provision of a multi-lingual capability within a DBMS is virtually an evolutionary imperative. From this, the name *multi-lingual database system* (MLDS) has been derived.

2. Some Benefits of MLDS

For any organization currently operating one or more of the existing model-based DBMSs, perhaps the most valuable feature of MLDS is the provision that permits existing databases to be "migrated" into MLDS. Following this migration of databases, data manipulations may be commenced in any or all of the supported languages: users comfortable with the "old" language may continue to execute familiar transactions, while all may experiment and explore the various operational features and capabilities offered by the several models and languages. In this way, a wider variety of both **types** and **modes** of transactions will be available. One need no longer be limited by a single model's *structure*, nor by the (perhaps limited) scope of its characteristic *transactions*.

At the same time, previously-written transactions need not be discarded with the advent of MLDS. Again, the multi-lingual environment ensures that the "old" language will be supported by MLDS, and an onerous and error-prone query/transaction conversion process need not be undertaken. Therefore, old transactions are given a reprieve, while new transactions--written in *any* of the supported languages--serve to augment and reinforce. We seemingly have achieved the best of both worlds, the old and the new.

Contrast the foregoing with an old-fashioned, single-language transfer; for example, suppose we had wanted to switch from the **network** to the **hierarchical** model. Not only would we be swapping one individual model and language for another (hoping to realize some net benefit in terms of representational accuracy and/or functionality), but the conversion process could become costly indeed. The structure of the database itself would require non-trivial alteration, together with the requirement to convert all existing transactions from the old to the new language.

A final advantage that we may mention that MLDS promises involves economy in the inevitable hardware upgrades. Eventually the time comes when, driven either by increasing system use or the irresistible attraction of technological improvements, it becomes necessary to upgrade the hardware capability. When separate systems exist (e.g., an hierarchical model-based system is maintained, together with a separate system supporting the network model), an upgrade will necessarily involve the hardware of *each* system, resulting in more effort and expense. Upgrading an MLDS, by way of contrast, will simultaneously benefit *all* users, regardless of their respective models of choice.

3. Enhanced Operational Characteristics

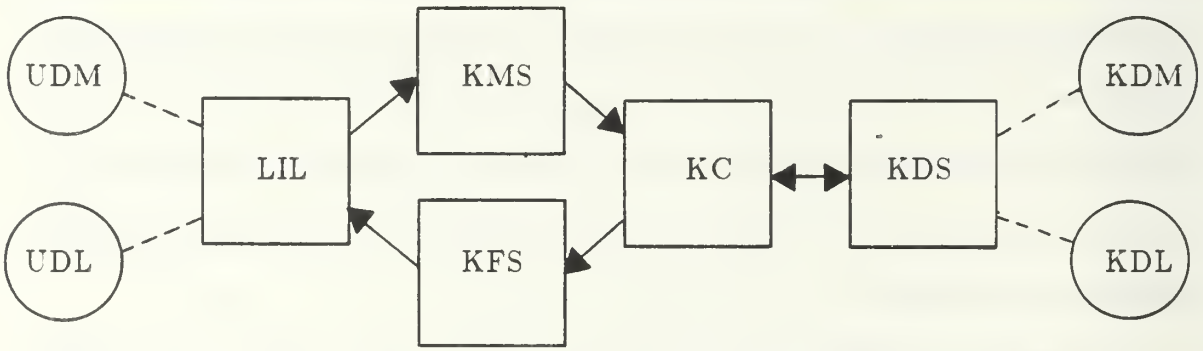
There are essentially two new operational characteristics that a user may exploit with MLDS. The first has already been alluded to: the capability of exploring and taking advantage of the strengths and the best features that each of the several supported models offers. A user need no longer be limited by one data model and language.

The second "enhancement" is the availability of the system's *native data model* and *data language*, known as the kernel data model (KDM) and the kernel data language (KDL), respectively. As implemented in MLDS, these are the attribute-based data model and the attribute-based data language. All other models may have their respective databases transformed into equivalent databases structured according to the kernel data model; additionally, each of the languages may have their various, respective transactions and queries translated into the kernel data language. In both cases, the transformation/translation is executed as a built-in facility of MLDS, requires no user-involvement, and thus is essentially transparent to the user.

However, KDM and KDL are more than simply the basic model/language; they are, in fact, high-level entities in the same sense as the other (supported) models and languages (e.g., the relational data model and SQL data language) are. The immediate result of this recognition is to provide the user with an *additional* model and language whose particular strengths and desirable traits are available for utilization.

4. The MLDS Structure and Functioning: An Overview

A high-level overview of the multi-lingual database system is provided in Figure 1. Utilizing a user-chosen data model (UDM), and writing transactions/queries in the corresponding user-chosen data language (UDL), the user interacts directly with the language interface layer (LIL). Transactions are then passed on to the kernel mapping system (KMS), which must perform two tasks: first, in the event the user has indicated an intention to create a new database, KMS transforms the supplied UDM database definition into an equivalent KDM definition, which is then sent to the kernel controller (KC). KC, in turn, forwards the KDM database definition to the kernel database system



UDM : User Data Model
UDL : User Data Language
LIL : Language Interface Layer
KMS : Kernel Mapping System
KFS : Kernel Formatting System
KC : Kernel Controller
KDS : Kernel Database System
KDM : Kernel Data Model
KDL : Kernel Data Language

Figure 1. The Multi-Lingual Database System.

(KDS). When this has completed, KDS notifies KC which then notifies the user (via the LIL) that the database definition has been processed, and that database loading may proceed.

The KMS's second task is to deal with UDL transactions. These transactions are translated by KMS into equivalent KDL transactions, and then forwarded to KC which, in turn, forwards them to KDS for actual execution. Following execution, the results are sent by KDS (in KDM-form) back to KC, which in turn sends these results to the kernel formatting system (KFS) for translation from KDM-form to UDM-form. Following this transformation, the "response set" is returned, again via the LIL, to the user.

It is important to note that the LIL, KMS, KC, and KFS together comprise the *language interface* of MLDS, and that a separate interface is required for each supported data model and language (see Figure 2). However, each of these interfaces shares the KDS. This arrangement, of course, reflects the natural, logical and efficient result of providing a system that affords a multi-lingual capability from the user's point of view, together with a single, unified data model and language from the system's point of view.

B. THE MULTI-BACKEND DATABASE SYSTEM (MBDS)

1. Background

Even as a mono-lingual database system presents severe limitations in database processing, so too does the traditional database system design which places the executing database system into main memory (which is shared by all users), while the databases themselves are placed throughout secondary storage

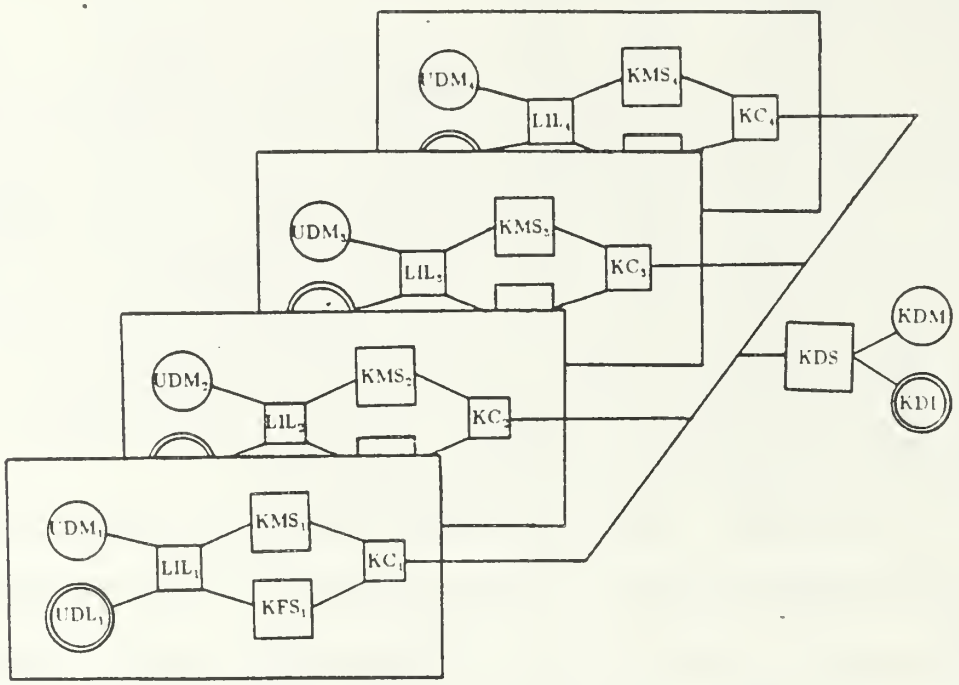


Figure 2. Multiple Language Interfaces for the Same Kernel Database System.

(which is, likewise, utilized by all users). Even comparatively low-intensity database system functioning may begin to encounter primary and secondary storage contention, thus increasing response times while simultaneously decreasing throughput. It is clear that database users' and non-database users' processing requirements may produce mutually unsatisfactory results.

As mentioned previously, this may in part be viewed as one result of utilizing a general-purpose computer that is, in fact, not even remotely optimized to perform database operations. As such operations increase, all users suffer more

or less equally, possibly bringing into play questions of equity: should everyone suffer for the "workloads of the few" (database users)?

The multi-backend database system (MBDS) is designed to solve both these performance, as well as the costly upgrade, problems (some of which we have already seen). The basic goal of the MBDS is to establish an extremely high-performance system for large-capacity databases. Within this primary goal are two distinct subgoals:

1. To observe a *reciprocal reduction* in the response times to user transactions as the number of "backends" in the MBDS is increased (while holding constant the database size, and the size of responses to transactions). We say in this case that we are seeking *performance gains* in terms of *response-time reductions.*

2. To observe *stable response times* to user transactions as the number of backends is increased proportionally to the increase of transaction responses. Here, we say that we are seeking *capacity growth* in terms of *response-time invariance* [Ref. 4: p. 9].

An additional goal of MBDS is that the system should be easily extensible, without any requirement for either new software or the modification of existing software. (Further information pertaining to the original design and analysis of MBDS may be found in [Ref. 5] and [Ref. 6].)

2. MBDS Configuration

The configuration of MBDS, as depicted in Figure 3, is actually quite straightforward. Utilizing essentially off-the-shelf hardware (together with rigorously specialized software--in fact, the MBDS has been described as a *software database solution* [Ref. 4: p. 8]), one microcomputer is designated as the *controller* while the remainder are the *backends.* Each backend has one or more

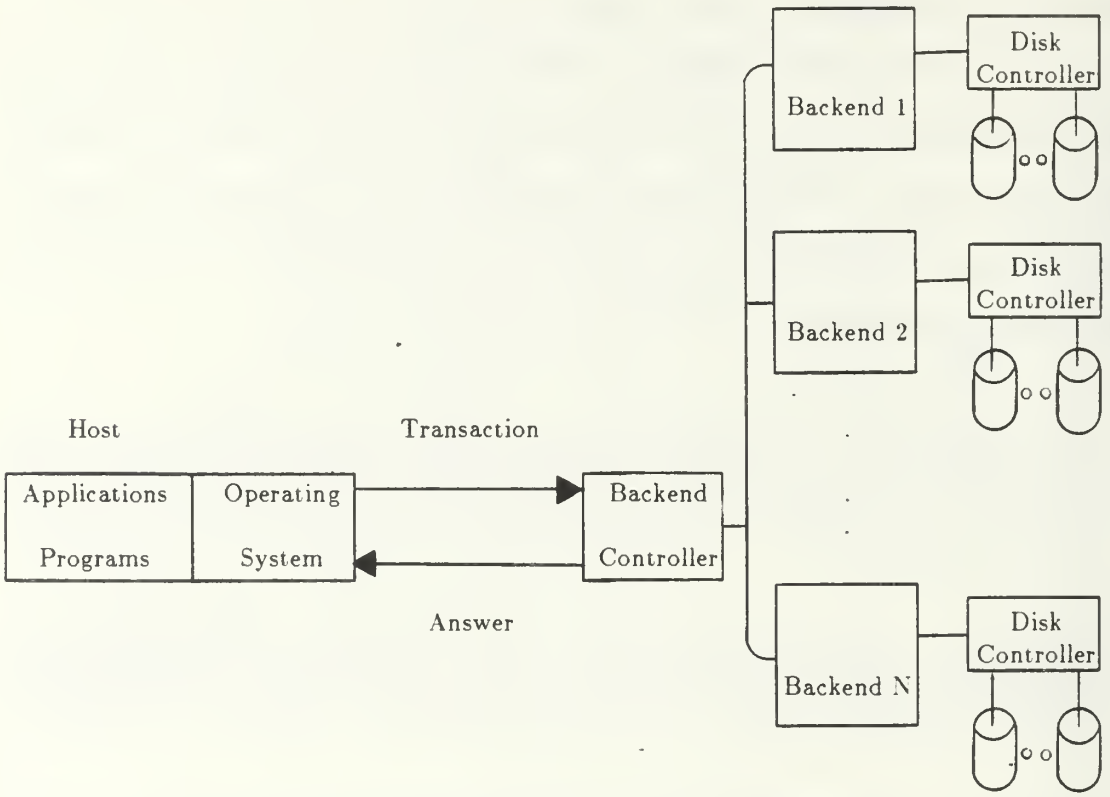


Figure 3. The Multi-Backend Database System.

dedicated disk drives, and the controller is attached to all backends by a broadcast bus.

As shown in Figure 3, virtually all database system functions have been downloaded from the host to the MBDS, leaving the applications programs to execute in the host. The database has likewise been downloaded, and distributed across the disk drives of the various backends. This serves greatly to facilitate parallel request processing.

At the Naval Postgraduate School Laboratory for Database Systems Research, two separate MBDSs are presently configured. The first consists of

eight ISI (Integrated Solutions. Inc.) workstations acting as backends. with a Digital Equipment Corporation VAX-11/750 acting as the controller. interconnected via Ethernet. The second. smaller configuration consists of two MicroVAX-IIs and a VAX-11/780, interconnected via DECNET. The basic ISI backend is equipped with a VME-68020 processor and a 2-MB main memory. together with 106-MB and 512-MB CDC Winchester disks. a VME-ED Ethernet board. and a VME-ICP8 communication board. The basic MicroVAX-II backends are configured with a Ka631 VAX-like processor on a Q22 bus and a 2-MB main memory, as well as a 71-MB RD53 fixed disk. a dual-drive 5-1/4" RX50 floppy disk. and a TK50 cartridge tape drive.

3. Operation of MBDS

The key to the improved performance of the MBDS rests chiefly with the parallelism achieved by the backends. However, with an arbitrary number of backends simultaneously performing their appointed tasks, the potential danger exists for the controller to become a bottleneck. This possibility has been minimized by giving the controller a minimum of functions to perform, placing the preponderance of the burden instead upon the backends.

Figure 4 gives a pictorial representation of the tasks of the controller. and of the backends. Beginning with the controller, we may observe that its responsibilities are described as *request preparation, insert information generation,* and *post processing.* **Request preparation** are those functions that must be attended to prior to broadcasting the request to the backends on the bus; these include parsing, syntax checking, and transforming the (validated) parsed requests into the forms the backends will require for their subsequent processing. **Insert information generation** functions pertain to an insert request. providing the

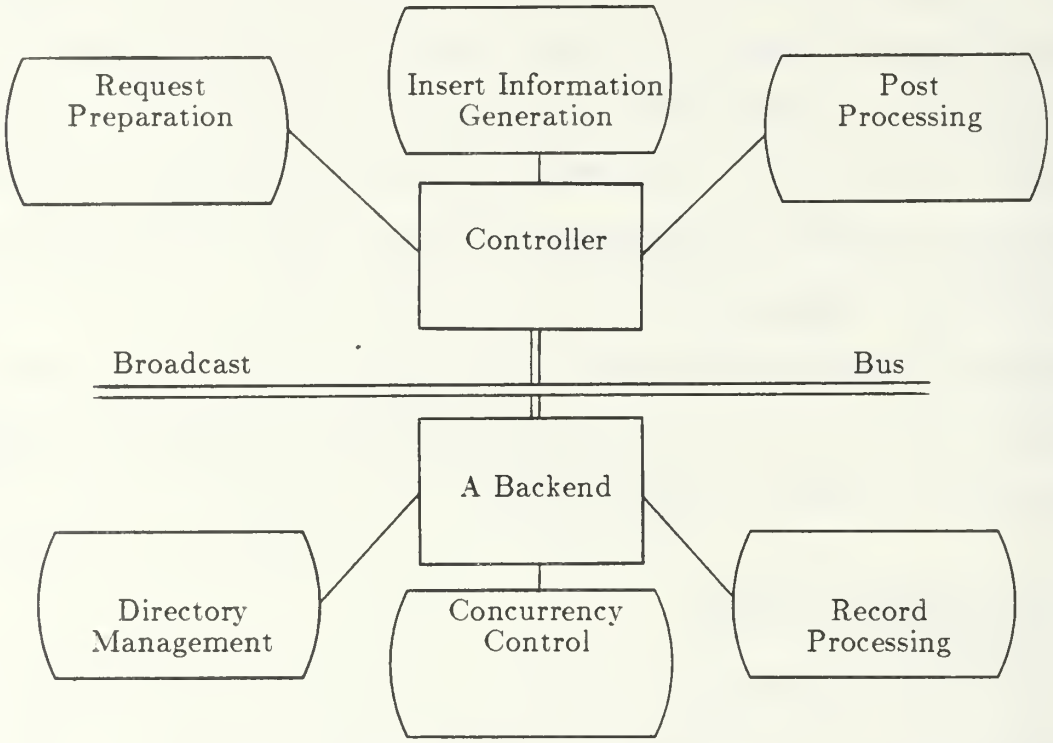


Figure 4. The MBDS Software Structure.

backends with such vital information as which backend is the proper destination for the record insertion. Finally, the **post processing** functions are those tasks--such as the collection of result data prior to forwarding to the host computer--that are appropriate once the backends have yielded up the request results.

The backends (each of which has identical software) perform the functions of *directory management*, *record processing*, and *concurrency control*. The **directory management** functions involve the determination of the addresses on secondary storage of identified records, ensuring that the directory table is maintained properly. **Record processing** functions are those that perform the store, select, and retrieve operations to/from secondary storage.

Concurrency control maintains consistency throughout the database despite the potentially conflicting demands of concurrently executing processes.

As previously noted, parallelism is the key to the improved performance that we seek. In addition, each backend maintains a queue of outstanding requests independently of all other backends. This serves to maximize the time spent in access operations, while simultaneously minimizing idle time. MBDS itself will in fact be capable of auto-configuring to any number of backends.

A final MBDS operational note that we should make is that users may access the system either through the host, or directly through the controller. (A more detailed description of MBDS may be found in [Ref. 7].)

C. THE ATTRIBUTE-BASED DATA MODEL (ABDM)

As we've seen above, while the user is free to work in the (supported) data model of choice with the multi-lingual database system, the multi-backend database system by contrast supports one language: the attribute-based data language (ABDL), based on the attribute-based data model (ABDM). Inasmuch as there are literally scores of reference resources pertaining to each of the supported models (the relational, network, and hierarchical models), we will not pursue them further here. Due to its lack of commercial realization, ABDM is not generally as well known; however, a few moments spent investigating it should prove to be a worthwhile investment.

The relevant constructs in the ABDM are the database, file, record, attribute-value pair, keyword, attribute-value range, directory keyword, non-directory keyword, directory, record body, keyword predicate, and query. Each of these is discussed in turn below.

Informally, a *database* consists of a collection of files. Each *file* contains a collection of records which are characterized by a unique set of keywords. A *record*, in turn, has two parts. The first of these is a collection of *attribute-value pairs* or *keywords*. An attribute-value pair is any given member of the Cartesian product of the attribute name and the value domain of the attribute. For example, <POPULATION, 25000> is an attribute-value pair in which the population attribute is paired with a value of 25000. Each record may contain at most one attribute-value pair for each attribute defined in the database. Following all attribute-value pairs is the second part of the record: textual information, which is referred to as the *record body*.

Certain attribute-value pairs of a record (or of a file) are defined to be the *directory keywords* of the record (file), because either the attribute-value pairs or their attribute-value ranges are kept in a *directory* for identifying the records (files). Conversely, those attribute-value pairs which are not kept in the directory are called *non-directory keywords*. An example of a record is shown below:

(<FILE, USCensus>, <CITY, Monterey>, <POPULATION, 25000>.
{Temperate climate})

Note that the angle brackets, <,>, enclose an attribute-value pair (keyword). The curly brackets, {,}, enclose the record body. The first attribute-value pair of all records in a given file will, by convention, be the same; specifically, the attribute will be FILE, and the value will be the file name. The entire record is enclosed in parentheses. The example record above is from the "USCensus" file.

Records in a database may be identified and/or accessed by means of keyword predicates. A *keyword predicate* is a 3-tuple consisting of a directory attribute, a relational operator (=, !=, <, >, ≤, ≥), and an attribute value. For

example. "POPULATION $\geq$ 20000" is a keyword predicate--specifically, a greater-than-or-equal-to predicate. A database *query* is formed by combining keyword predicates into a form known as disjunctive normal form. For example, the query

(FILE = USCensus and CITY = Monterey) or
(FILE = USCensus and CITY = San Jose)

would be satisfied by all records in the USCensus file with a CITY value of either Monterey or San Jose. Parentheses bracketing the conjunctions of a query are used to ensure clarity and avoid any potential ambiguity. For additional information pertaining to the attribute-based data model, the interested reader is directed to [Ref. 3: pp. 9-10].

III. THE DATA MODEL TRANSFORMATIONS

The purpose of this chapter is to investigate the various techniques used to translate the database descriptions from the respective (supported) data model descriptions to a description amenable to the multi-backend database system (MBDS). Following this, we review some of the actual, dynamic data structures used in the multi-lingual database system (MLDS) to represent the relational, hierarchical, and network models.

As we saw in the previous chapter, MLDS provides the extremely valuable service of permitting the user to establish databases and perform queries/transactions in the (supported) data model and language of choice. Since MBDS is capable only of dealing with databases and queries/transactions structured according to the KDM (kernel data model) and the KDL (kernel data language)--i.e., the ABDM (attribute-based data model) and the ABDL (attribute-based data language) respectively--the multi-lingual capability must be supported by *transforming* the user-selected database model into ABDM and *translating* the user transactions of a model-based language into ABDL.

This thesis is concerned with the transformation of data models. In the data-model transformation process, it is essential to preserve the data semantics of the user data model (UDM), together with ensuring operational equivalence. The strength and power of the kernel data model and language are such that the transformation process *preserves the semantics* of UDM, and that the required operational equivalence *is* attained. Thus, for our purposes, the data semantics of

the relational model, of the network model, and of the hierarchical model are all successfully preserved within the attribute-based model.

This thesis is *not* concerned with the translation of queries and transactions; interested readers are directed to [Ref. 8] for the implementation of a Network language interface, to [Ref. 9] for the implementation of a Relational language interface, and to [Ref. 10] for the implementation of an Hierarchical language interface.

A. THE MODEL-SPECIFIC TRANSFORMATIONS

1. The Relational-to-Attribute-Based Transformation.

Recall that relational data is organized into *tuples* of *relations*. A *database* is a collection of relations. The tuples of a relation have the property that no two tuples may be identical; furthermore, the *attributes* of a relation must all be distinct. In Figure 5, we define the Course-Prereq-Offering database in the relational format; we will continue to utilize this basic database in subsequent examples.

Course(Course#, Title, Descrip)
Prereq(Course#, Pcourse)
Offering(Date, Location, Format)
Schedule(Course#, Date)

Figure 5. A Relational Version of the Course-Prereq-Offering Database.

Let us briefly describe the relations of this database. In the Course relation, we note that each course is uniquely defined by a course number, and that each course additionally has a title and a description. The Prereq relation indicates that a course may have one or more prerequisites, while the Offering relation uniquely specifies date, location, and format for courses offered. Finally, the Schedule relation indicates those courses that are offered on one or more dates.

The relational-to-attribute-based mapping is the most straightforward and uncomplicated of those that will be discussed in this thesis. A conceptual mapping shows that the database, the relation, and the tuple in the relational model correspond directly to the database, the file, and the record, respectively, in the attribute-based model. Thus, the relational database depicted in Figure 5 is mapped to the attribute-based database in Figure 6.

As can readily be seen in Figure 6, each relational attribute has been mapped into an attribute-based keyword. The place-holder "value" has been inserted above to represent the value-portion of the attribute-value pair.

(<FILE, Course>, <COURSE#, value>, <TITLE, value>,
<DESCRIP, value>)
(<FILE, Prereq>, <COURSE#, value>, <PCOURSE#, value>)
(<FILE, Offering>, <DATE, value>, <LOCATION, value>,
<FORMAT, value>)
(<FILE, Schedule>, <COURSE#, value>, <DATE, value>)

Figure 6. An Attribute-Based Mapping of the Relational Course-Prereq-Offering Database.

Additionally, a keyword whose attribute is FILE has been included in each record, and whose place-holder has the relation name.

The specific transformational algorithm for this, and for the other data models, will be given in Chapter IV.

2. The Hierarchical-to-Attribute-Based Transformation.

Unlike relational data, hierarchical data is organized into *occurrences* of *segments*. A *database* therefore consists of a collection of segments which are structured in an hierarchical (tree-like) fashion. As has been characteristic of the tuples in a relation, no two occurrences of a segment may be identical. Likewise, the *fields* of an occurrence are distinct, as have been the attributes of a relation. Finally, we may characterize a parent-child relationship between segments in which an occurrence of one segment (the parent) may correspond to one or more occurrences of another segment (the child). Figure 7 provides a pictorial representation of the Course-Prereq-Offering hierarchical database that is equivalent to the relational database of Figure 5.

We may note that an hierarchical database, by definition, has *levels*, where the *root* (uppermost) segment is defined to be at level 0, children of the root at level 1, grandchildren of the root at level 2, and so forth. In Figure 7, the Course segment is the root of the hierarchy, and uniquely identifies a course by Course#; course titles and descriptions are additionally provided. The children of the Course (root) segment are the Prereq and Offering segments. Thus, each course may have one or more prerequisites, and may have one or more offerings--both of which represent one-to-many relationships.

In the hierarchical-to-attribute-based mapping, the hierarchical notions of database, segment, and occurrence are mirrored by the attribute-based

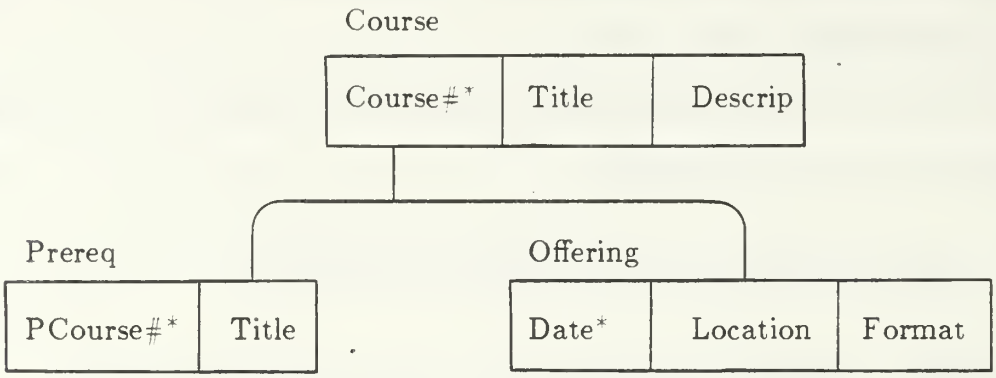


Figure 7. An Hierarchical Version of the Course-Prereq-Offering Database.

concepts of database, file, and record, respectively. An essential aspect of this mapping, however, is to preserve the semantics of the one-to-many relationships present in the hierarchy, by including in the attribute-based record the sequence fields along the hierarchical path *from* the root segment *to* the current segment, and all of the segments between them. Figure 8 is therefore the attribute-based mapping of our hierarchical database.

As we may quickly discern in Figure 8, each field of a segment occurrence has been transformed into a keyword of a record. Again, the place-holder "value" has been employed to represent the *value* portion of an *attribute-value* pair. Reminiscent of the relational-to-attribute-based transformational technique, the segment name has been included as an attribute value in the FILE keyword. Finally, we note that the Course sequence field--Course#--has been *cascaded* into the attribute-based record types for the children records, Prereq and Offering. As described above, this action will preserve the one-to-many

(<FILE, Course>, <COURSE#, value>, <TITLE, value>,
<DESCRIP, value>)
(<FILE, Prereq>, <COURSE#, value>, <PCOURSE#, value>,
<TITLE, value>)
(<FILE, Offering>, <COURSE#, value>, <DATE, value>,
<LOCATION, value>, <FORMAT, value>)

Figure 8. An Attribute-Based Mapping of the Hierarchical Course-Prereq-Offering Database.

relationship between the Course segment and the Prereq segment, and between the Course segment and the Offering segment.

It is interesting to note that if the Offering segment, for example, had had one or more children, the cascading effect would become even more apparent: each such child segment would include (in addition to the FILE keyword defining the segment name, and the keywords corresponding to each field of the segment) a keyword for *each* sequence field in *each* segment between the current segment and the root, inclusively. In other words, the child segment would additionally have a Date keyword, and a Course# keyword. In this way, we may implicitly observe that the cascading action necessary to preserve the successive one-to-many relationships throughout the database will result in *longer* attribute-based files the *lower* the original hierarchical segment is in the hierarchy.

3. The Network-to-Attribute-Based Transformation.

The network data model is closely associated with the concept of the digraph, or *directed graph*. The nodes of the graph represent the *records* of the network, while the arcs represent the *relationships* between records. The principal

distinction between the network and the hierarchical models lies in the fact that, while one-to-many relationships may exist in the hierarchical model, the more general many-to-many relationships may be found in the network model.

In order to represent the many-to-many relationship, the concept of the *set* is introduced. A set is a relationship between record types in which a single record type is determined to be the set *owner*, and another record type is denoted the *member* record type. Of special note in this regard is the prohibition that an owner record type may not be a member of the same set; this CODASYL requirement eliminates the possibility of recursive definition within a set. By strategically organizing into two separate one-to-many sets, a many-to-many structure may be represented hierarchically. Figure 9 is a graphic representation of the Course-Prereq-Offering network database that is equivalent to the relational database of Figure 5.

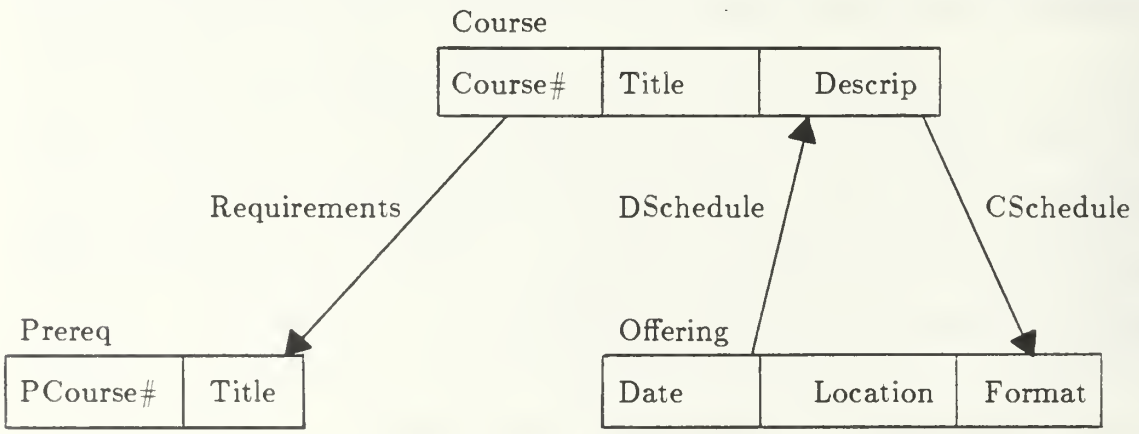


Figure 9. A Network Version of the Course-Prereq-Offering Database.

In Figure 9, the arcs point *from* the set owner *to* the set member of a given set (e.g., the "Requirements" set). Thus, we see that we have achieved a many-to-many relationship between the record types Course and Offering, by establishing two separate one-to-many relationships (sets) between the respective record types. Figure 9 therefore informs us that a given Course may have one or more Offerings; at the same time, a given Offering (i.e., a Date, Location, and Format) may be associated with one or more Courses.

In establishing a network-to-attribute-based mapping, we observe that the network concepts of database, record, occurrence, and data item are represented in the corresponding attribute-based notions of database, file, record, and attribute, respectively. As in the previous model-specific examples, attribute-based keywords will be constructed from the elementary data items. Additionally, each record occurrence of the network database must belong to a particular type which, again, will be distinguished by the value of the attribute FILE. Both of these conventions are consistent with the foregoing relational and hierarchical practices.

Several new concepts must be introduced, however, to fully capture the distinctive flavor of a network database model. The first of these reflects the requirement that each record occurrence of a network database have a unique database key (or address) associated with it. This key (address) is assigned by the language interface, and is transparent to the user. The key may be represented by the following attribute-based form, where DBKEY is a literal:

<DBKEY, key-value>

Next, information must be generated that clearly identifies network set membership, and within a specified set, the set ordering. Inasmuch as occurrences

of set types are "pairwise disjoint"--a record occurrence may not be present in two different occurrences of the same set type--each member record occurrence belonging to a set occurrence is thus also uniquely identified by its owner record occurrence. Set membership can therefore be expressed by inclusion of the keyword

<MEM.set-name. owner-key-value>

for each set occurrence in which the record is a member.

Finally, it is often useful to know the logical position of a record occurrence within a specified set occurrence. This may be accomplished by including the keyword

<POS.set-name. sequence-value>

in the attribute-based record for each set of which the record is a member.

With this information, we are now in a position to express a complete mapping from the network to the attribute-based model of our Course-Prereq-Offering database, as shown in Figure 10.

From Figure 10 it may clearly be seen that each network data item has been mapped to a keyword; the record type has likewise been mapped to a keyword whose attribute name is the constant "FILE." Once again, the place-holder "value" is meant to represent the *value* portion of the *attribute-value* pair (keyword). The additional keyword of "DBKEY," together with its place-holder "key-value," associates the key (or address) with the record occurrence. Meanwhile, "MEM" and "POS" convey essential information pertaining to the set membership of the record instance. The second argument in both of these keywords identifies the set of which the record is a member. The place-holder

(<FILE, Course>, <DBKEY, key-value>, <COURSE#, value>,
<TITLE, value>, <DESCRIP, value>,
<MEM.DSchedule, owner-key-value>,
<POS.DSchedule, sequence-value>),
(<FILE, Prereq>, <DBKEY, key-value>, <PCOURSE#, value>,
<TITLE, value>,
<MEM.Requirements, owner-key-value>,
<POS.Requirements, sequence-value>),
(<FILE, Offering>, <DBKEY, key-value>, <DATE, value>,
<LOCATION, value>, <FORMAT, value>,
<MEM.CSchedule, owner-key-value>,
<POS.CSchedule, sequence-value>)

Figure 10. An Attribute-Based Mapping of the Network Course-Prereq-Offering Database.

"owner-key-value" uniquely identifies the set owner, while the place-holder "sequence-value" provides for the ordering of record occurrences within the set occurrence.

B. THE DYNAMIC STRUCTURES

Recall that the user interacts with the *language interface* of the multi-lingual database system (MLDS) in the (supported) model and language of choice. These are designated as the user data model (UDM) and the user data language (UDL), respectively (as depicted in Figure 1). During the process of describing and constructing a database configured according to the chosen model, a database schema definition is entered in the syntax of that model. An example schema

adapted from [Ref. 8: p. 36]. utilizing the network model and partially representing our example database. is shown in Figure 11.

It is important for MLDS to maintain and have available the user's schema. All user interaction will be based on the specific user's schema, and the language interface is charged with validating all transactions against the schema prior to their translation into the kernel language. Furthermore. this schema serves initially as the basis for the required model transformation to the attribute-based data model (ABDM).

```
SCHEMA NAME IS SCHOOL-DAYS.
  RECORD NAME IS COURSE:
    DUPLICATES ARE NOT ALLOWED FOR COURSE#.
        COURSE#     ; TYPE IS CHARACTER 6.
        TITLE       ; TYPE IS CHARACTER 20.
        DESCRIP     ; TYPE IS CHARACTER 25.

  RECORD NAME IS PREREQ;
        PCOURSE#    ; TYPE IS CHARACTER 6.
        TITLE       ; TYPE IS CHARACTER 20.

  RECORD NAME IS OFFERING:
        DATE        ; TYPE IS FIXED 6.
        LOCATION    ; TYPE IS CHARACTER 15.
        FORMAT      ; TYPE IS CHARACTER 5.

  SET NAME IS REQUIREMENTS:
    OWNER IS COURSE:
        ORDER IS SORTED BY DEFINED KEYS
        DUPLICATES ARE NOT ALLOWED.
    MEMBER IS PREREQ:
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        KEY IS ASCENDING PCOURSE# IN PREREQ;
        SET SELECTION IS BY-VALUE OF COURSE# IN COURSE.
```

Figure 11. Network Database Schema.

For these reasons, dynamic data structures are generated from the user-entered database schema. The next several figures will serve to illustrate the portions of those structures relevant to this thesis.

1. The Relational Structure.

Relevant portions of the relational schema are shown in Figure 12. Readers interested in further exploring the dynamic structures created by this schema are directed to [Ref. 9].

For our purposes, there are three principal structures of concern in the relational schema: the rel-dbid-node, the rel-node, and the rattr-node. Beginning with the first of these, the rel-dbid-node (for RELational DataBase IDentification NODE) provides information about the relational database in general: the

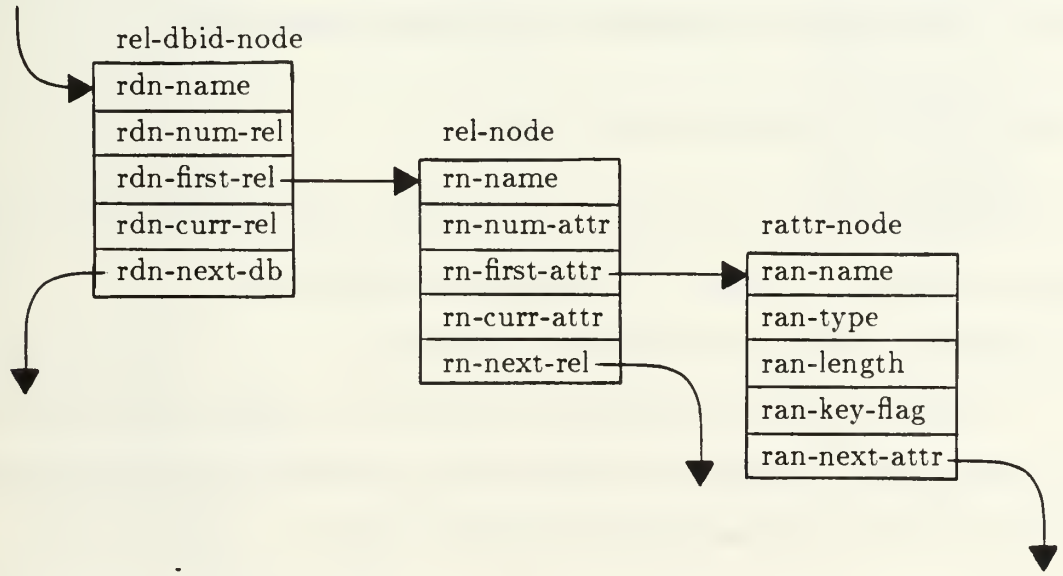


Figure 12. Relational Database Schema Data Structures.

database name (rdn-name), the number of relations comprising the database (rdn-num-rel), together with pointers to the first relation in the schema (rdn-first-rel), the relation currently being accessed (rdn-curr-rel), and the next defined database (rdn-next-db).

The rel-node (for RELation NODE) provides similar information pertaining to a relation node: the relation name (rn-name), the number of attributes in the relation (rn-num-attr), and pointers to the first attribute of the relation (rn-first-attr), the attribute currently being accessed (rn-curr-attr), and the next relation in the database (rn-next-rel).

The final node of interest is the rattr-node (for Relational ATTRibute NODE), which identifies elements pertaining to the attributes of a relation: the attribute name (ran-name), the attribute type--either f(loat), i(nteger), or s(tring)--(ran-type), the maximum length of the attribute value (ran-length), a flag indicating whether the attribute has been designated a key (ran-key-flag), and finally a pointer to the next attribute in the relation (ran-next-attr).

2. The Hierarchical Structure.

Slightly more complicated, the relevant portions of the hierarchical schema are shown in Figure 13. Interested readers are directed to [Ref. 10] for a more complete rendering of these dynamic structures.

Again, there are three principal structures of interest; however, in this case, three separate instances of the hrec-node have been shown in Figure 13, as an aid in discerning the hierarchical flavor of the schema.

To summarize the information contained in these nodes, we begin with the hie-dbid-node (for HIErarchical DataBase IDentification NODE). This structure contains the database name (hdn-name), the number of segments in the

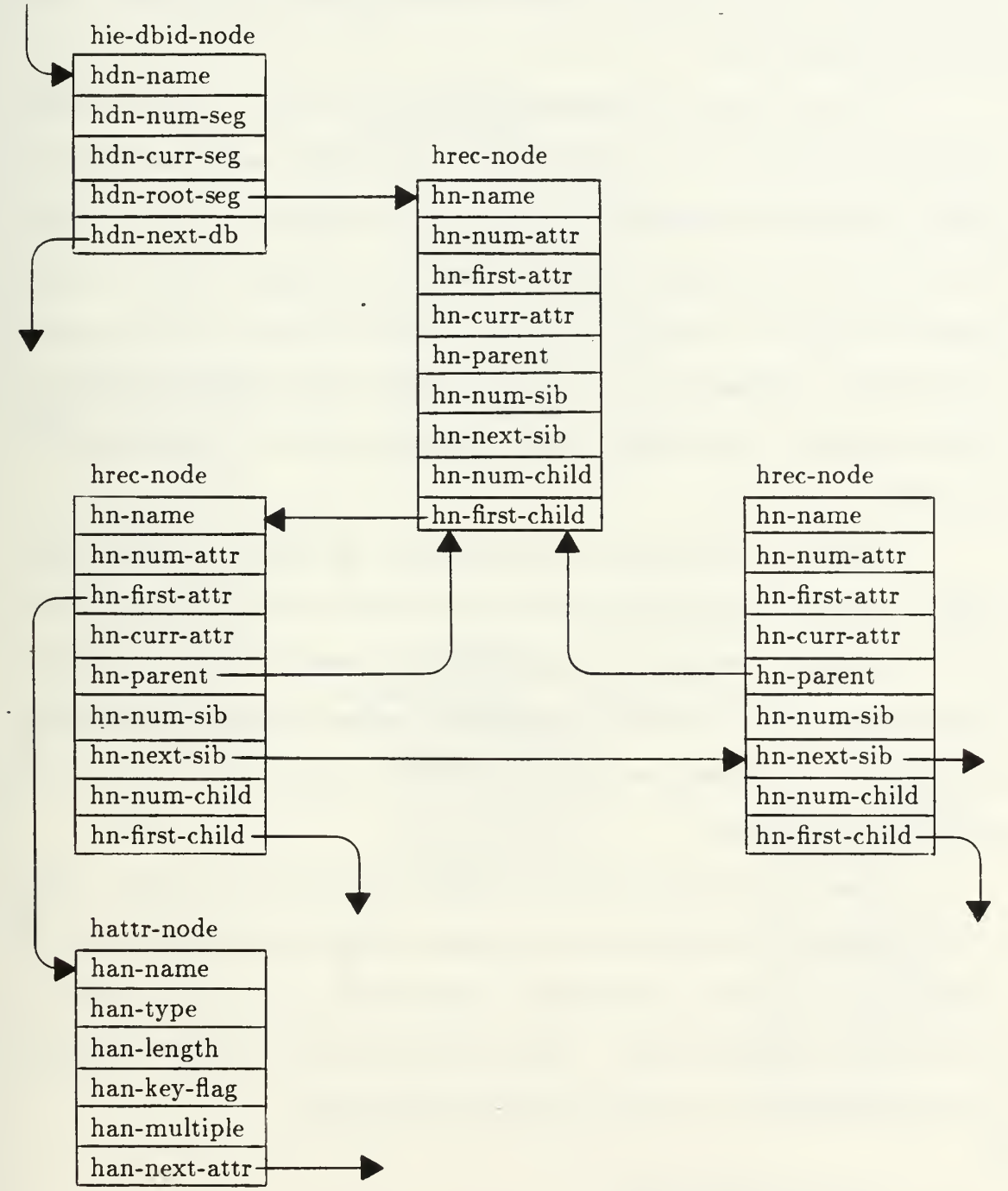


Figure 13. Hierarchical Database Schema Structures.

database (hdn-num-seg), together with pointers to the segment currently being accessed (hdn-curr-seg), the root segment (hdn-root-seg), and the next defined database (hdn-next-db).

The hrec-node (for Hierarchical RECord--actually, segment--NODE) contains a great many parts. These include the segment name (hn-name), the number of fields in the segment (hn-num-attr), and pointers to the first field (hn-first-attr), the field currently being accessed (hn-curr-attr), and the parent segment (hn-parent). Additional elements include the number of sibling segments (hn-num-sib), a pointer to the next sibling segment (hn-next-sib), the number of children segments (hn-num-child), and a pointer to the first child segment (hn-first-child).

Finally, the hattr-node (for Hierarchical ATTRibute--actually, field--NODE) consists of the field name (han-name), the field type (han-type), the field length (han-length), together with a flag indicating whether the field is a key field (han-key-flag), a flag to indicate whether twin segment occurrences of this type may contain the same sequence field values (han-multiple), and a pointer to the next field (han-next-attr).

As may be apparent from the foregoing, the naming conventions utilized in these data structures may not in every instance result in the ideal identifiers for model-specific constructs. For example, "han-next-attr" is used, rather than the more expressive "hfn-next-field." This was done, however, to achieve uniformity *across* data models, at the expense of model-specific clarity.

3. The Network Structure.

Finally, we are ready to deal with the relevant parts of the network schema, shown in Figure 14. There are four structures of interest to us here.

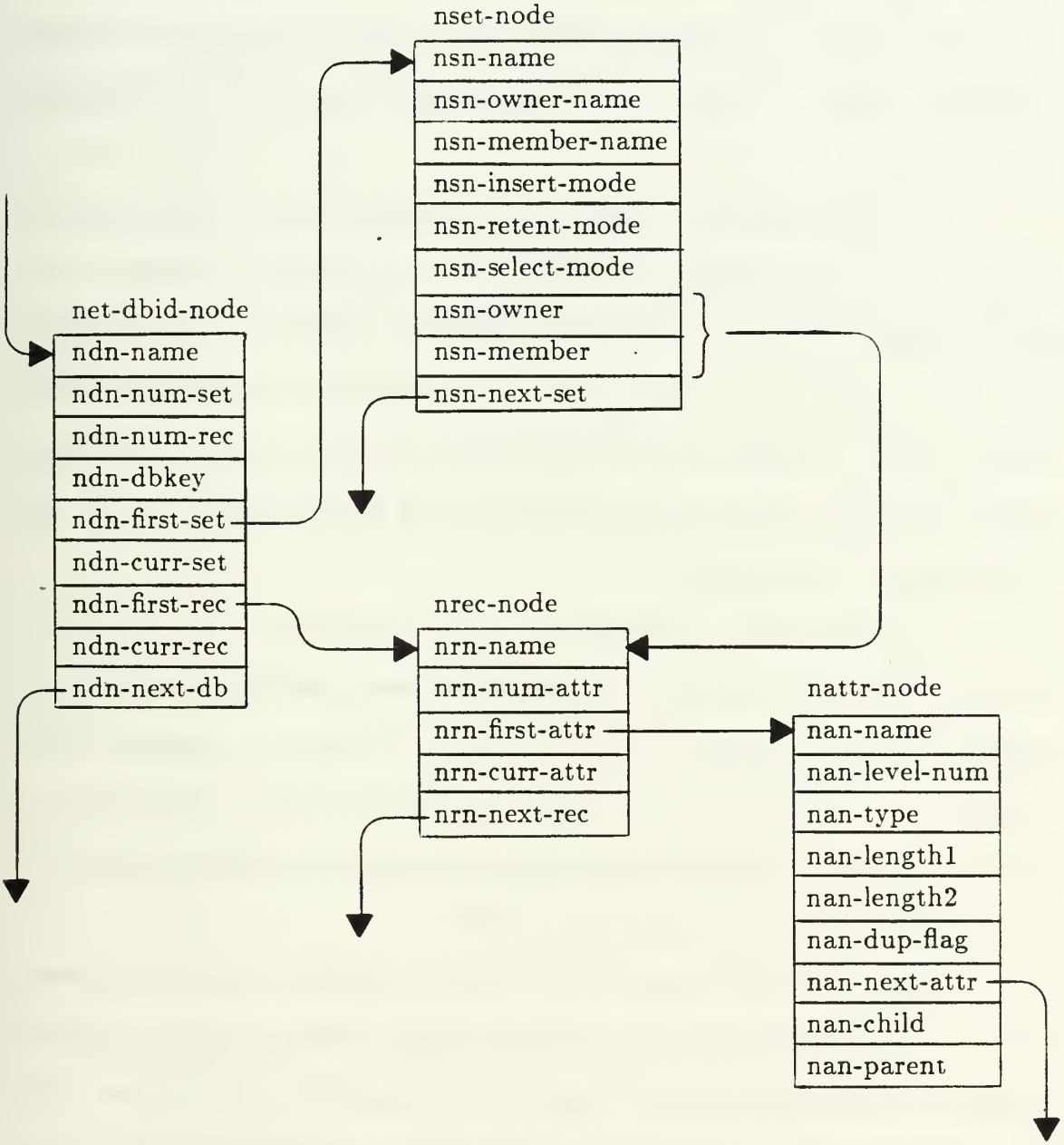


Figure 14. Network Database Schema Structures.

The network structures are the most complicated of those we have dealt with thus far. The four structures of interest are the net-dbid-node (for NETwork DataBase IDentification NODE), the nset-node (for Network SET NODE), the nrec-node (for Network RECord NODE), and the nattr-node (for Network ATTRibute NODE). Again, a brief description of their respective constituent parts is in order.

The net-dbid-node consists of the database name (ndn-name), the number of sets in the database (ndn-num-set), the number of records in the database (ndn-num-rec), and the next database key value to use when visiting new records (ndn-dbkey). Additionally, there are pointers to the first set (ndn-first-set), the set currently accessed (ndn-curr-set), the first record in the database (ndn-first-rec), the record currently being accessed (ndn-curr-rec), and the next defined database (ndn-next-db).

The nset-node is concerned with a given network set. It consists of the set name (nsn-name), the name of the set owner (nsn-owner-name) and of the set member (nsn-member-name), and the mode of insertion (nsn-insert-mode), retention (nsn-retent-mode), and selection (nsn-select-mode). Additionally, it contains pointers to the set owner (nsn-owner) and the set member (nsn-member), as well as to the next network set (nsn-next-set).

The nrec-node pertains to the individual network records. Information contained in this node includes the record name (nrn-name), the number of attributes contained in the record (nrn-num-attr), together with pointers to the first of these attributes (nrn-first-attr), the attribute currently accessed (nrn-curr-attr), and the next record in the database (nrn-next-rec).

Finally, the nattr-node is a structure containing information about the individual attributes that make up a record. The pertinent data contained in the

nattr-node includes the attribute name (nan-name), the level number of this attribute (nan-level-num), the attribute type (nan-type), as well as the maximum length that a value of this attribute may possibly have (nan-length1), the maximum length of the decimal portion of this attribute (nan-length2), and a flag which denotes whether duplicates are allowed (nan-dup-flag). Lastly, pointers to the next attribute for the current record (nan-next-attr), a "child" attribute (nan-child), and the "parent" attribute (nan-parent) complete the list.

In the next chapter, we will see how each of these data structures is used to construct the Template and Descriptor files required by MBDS to implement the kernel model.

IV. THE INTERFACE IMPLEMENTATION

A. AN OVERVIEW

In the preceding chapters, we have discussed many of the interface issues pertaining to the multi-lingual database system (MLDS) and the multi-backend database system (MBDS), in the abstract. This has been done to provide an overview of the interface concepts, without overwhelming the reader with an abundance of implementation details.

The present chapter is, in a very real sense, the *essence* of this thesis. The concepts and structures presented here, together with the respective data structures presented in the latter part of the preceding chapter, form the essential interface implementation.

There are two essential and separate portions to this interface. The first consists of a transformation that defines the user's database to the MBDS. The appropriate dynamic structures presented in the preceding chapter are utilized to perform this transformation, and a special file, called the Template File, is created. It is by means of this file that MBDS will be able to recognize and manipulate the user's database.

The second phase of the interface involves the creation of the database indices. This phase differs from the first phase in that, while the creation of the Template File is accomplished without the user's knowledge or participation, the creation of indices necessarily involves the user. It is, in fact, the user's detailed knowledge of the database and intended uses of the database that will permit an

effective choice of database indices, which is stored in a Descriptor File. On the basis of the chosen indices, MBDS then automatically forms the clusters for the user, although the concept of "clusters" is not necessarily known to the user. Active user participation is thus required in the creation of the Descriptor File, which MBDS will subsequently use in the structuring of the database into clusters.

Each of these phases will thus be taken up in turn, beginning with the creation of the Template File, and completing with the establishment of the Descriptor File. There are three procedures that have been written for each of the two phases of the implementation, effecting the transformations of the relational, hierarchical, and network models. References are made to the appropriate Appendices, where the algorithms defined in this chapter are programmed in the C programming language to create the required files.

B. CREATING THE TEMPLATE FILES

A very specific structure is required of the Template File; Figure 15 indicates its syntax. The syntax shown is described in attribute-based terms; appropriate translations should be made, as necessary, for the relational, hierarchical and network models.

In order to make this abstract structure more understandable, Figure 16 is the Template File created from the relational database of Figure 5.

1. The Relational Algorithm

Now we are in a position to write the required transformational algorithms. As usual, we begin with the relational model. Figure 17 represents

```
DATABASE-NAME
Number-Of-Files
Number-Of-Attributes
    -In-First-File                  (Add one for the Constant attribute "FILE")
Name-Of-First-File
FILE s                              (s(tring), i(nteger), or f(loat))
First-Attribute    Attribute-Type
Second-Attribute   Attribute-Type
<. . . . .>
i'th-Attribute     Attribute-Type
Number-Of-Attributes
    -In-Second-File                 (Add one for the Constant attribute "FILE")
Name-Of-Second-File
FILE s
First-Attribute    Attribute-Type
Second-Attribute   Attribute-Type
<. . . . .>
j'th-Attribute     Attribute-Type
<. . . . .>
<. . . . .>
Last-Attribute-Of
    -Last-File  Attribute-Type
```

Figure 15. The Template-File Syntax.

an algorithm to transform the data structures of Figure 12 into the Template File structure of Figure 15. Appendix A is the C-procedure that implements this algorithm.

Due to the relatively linear structure of the relational database structures (as seen in Figure 12), the algorithm of Figure 17 is clean and spare. The

```
SCHOOL-DAYS          (database name)
4                    (number of relations)
4                    (attributes in 1st relation, including "FILE")
COURSE               (name of 1st relation)
FILE s               (constant attribute)
COURSE# s
TITLE s
DESCRIP s
3                    (attributes in 2nd relation)
PREREQ               (name of 2nd relation)
FILE s
COURSE# s
PCOURSE s
4                    (attributes in 3rd relation)
OFFERING             (name of 3rd relation)
FILE s
DATE i
LOCATION s
FORMAT s
3                    (attributes in 4th relation)
SCHEDULE             (name of 4th relation)
FILE s
COURSE# s
DATE i
```

Figure 16. A Relational-Database Template.

procedure of Appendix A has the comparatively simple task of traversing a sequence of linked lists, withdrawing the required information (e.g., the database name, number of relations, and so forth), and writing this information to the

Assertions:

1. Relational Database D has relations $\{R_1, R_2, \ldots R_n\}$.
2. Each relation R_i, i = 1.....n. has the relation name R_i-name.
3. Each relation R_i, i = 1,....n, has A_{R_i} attributes.
4. Each attribute $A_{i,j}$, j = 1,.....A_{R_i}, has attribute name $A_{i,j}$-name.
5. Each attribute $A_{i,j}$, j = 1,...,A_{R_i}, has attribute type $A_{i,j}$-type.

Algorithm:

```
write D-Name        /* Database Name */
write n             /* Number of Relations */

/* Repeat for each relation in database */
for each relation R_i in database D do
    {
    write A_Ri + 1          /* Add 1 for "FILE" */
    write R_i-name          /* Relation Name */
    write "FILE s"

    /* Repeat for each attribute in relation */
    for each attribute A_i,j in relation R_i do
        {
        write A_i,j-name  A_i,j-type     /* Attribute name. type */
        }
    }
```

Figure 17. Relational Template-File-Transformation Algorithm.

Template File. Succeeding procedures find a somewhat more demanding task, due in large part to the rather more complicated structures of the hierarchical and network data structures.

2. The Hierarchical Algorithm

Next, we turn our attention to the hierarchical schema. Recall from Chapter III that, first of all, a change of terminology is in order: "segments" replace "relations," and "fields" are used instead of "attributes." Additionally, the straightforward linked-list traversal that we have found so convenient in the relational schema cannot be used here. Instead, we have a requirement to "cascade" sequence fields into the child segments of the hierarchy from the parent segments. Keeping these things in mind, we now turn to the Hierarchical Transformation Algorithm, given in Figure 18.

Appendix B is a realization of the algorithm of Figure 18, written as a C-procedure. Again, this algorithm is very high-level, and does *not* address all of the details that an actual program must deal with. For example, the traversal of the hierarchy is accomplished by a pre-order recursive traversal technique in Appendix B. Also, "saving" the cascading sequence fields is done by building a (temporary) linked list from these fields; we may then simply traverse this list when it comes time to write these fields to the file for the instant segment.

3. The Network Algorithm

Finally, we are ready to address the Network algorithm, given in Figure 19. Note in Figure 19 that the character "^" is used to indicate concatenation. There, the concatenation of either "MEM" or "POS" to the set name requires no intervening spaces. For example, if a record happens to be a member of the set "REQUIREMENTS," then we write the following two lines to the file:

Assertions:

1. Hierarchical Database D has segments $\{S_1, S_2, \ldots, S_n\}$.
2. Each segment S_i, i = 1,....,n, has segment name S_i-name.
3. Each segment S_i, i = 1,....,n, has F_{S_i} fields.
4. Each field $F_{i,j}$, j = 1,....F_{S_i}, has field name $F_{i,j}$-name.
5. Each field $F_{i,j}$, j = 1,....,F_{S_i}, has field type $F_{i,j}$-type.

Algorithm:

```
write D-name            /* Database Name */
write n                 /* Number of Segments */
/* Repeat for each Segment in the Database */
for each segment S_i in database D do
    {
    Trace path back to root segment
    /* Repeat for each intermediate Segment */
    for each segment on path to root do
        {
        count sequence fields, F_X_i
        save sequence fields, F_X_i
        }
    write (F_S_i + F_X_i + 1)      /* Add 1 for "FILE" */
    write S_i-name
    write "FILE s"
    /* Repeat for each field in segment */
    for each field F_i,j in segment S_i do
        {
        write F_i,j-name   F_i,j-type      /* Field name, type */
        }
    /* Repeat for each cascaded field saved above */
    for each saved cascaded sequence field F_X_i do
        {
        write F_X_i-name   F_X_i-type
        }
    }
```

Figure 18. The Hierarchical Template-File-Transformation Algorithm.

Assertions:

1. Network Database D has records $\{R_1, R_2, \ldots R_n\}$.
2. Each record R_i, i = 1.....n, has the record name R_i-name.
3. Each record R_i, i = 1.....n, has A_{R_i} attributes.
4. Each attribute $A_{i,j}$, j = 1,....A_{R_i}, has attribute name $A_{i,j}$-name.
5. Each attribute $A_{i,j}$, j = 1,...,A_{R_i}, has attribute type $A_{i,j}$-type.
6. Each set S_k, k = 1,....,m has set name S_k-name.

Algorithm:

```
write D-name            /* Database Name */
write n                 /* Number of Relations */

/* Repeat for each record in database */
for each record R_i in database D do
    {
    determine how many sets record is member of, "x"
    write (A_Ri + 2x + 2)      /* Number of Attributes */
    write R_i-name
    write "FILE s"
    write "DBKEY i"

    /* Repeat for each attribute in record */
    for each attribute A_i,j in record R_i do
        {
        write A_i,j-name   A_i,j-type     /* Attribute name. type */
        }

    /* Repeat for each set record is member of */
    for each set record is member of do
        {
        write MEM^S_k-name  "i"
        write POS^S_k-name  "i"
        }
    }
```

Figure 19. The Network Template-File-Transformation Algorithm.

MEMREQUIREMENTS i

POSREQUIREMENTS i

Appendix C contains the C-procedure written for the algorithm of Figure 19. Because of the layout of the network data structure (see Figure 14), a recursive routine is required to traverse the attributes of a given record.

As discussed in Chapter III. we have--in addition to the "File" attribute, written for *all* schemas--the attribute "DBKEY," and the attributes "MEM"^S_k-name and "POS"^S_k-name for each set of which a given record is a member. Of course, the attributes of the record are written out in the usual manner.

A challenging aspect of the Network Template is the determination of the number of attributes for a given record. This is shown in the algorithm as

$$(A_{R_i} + 2x + 2)$$

where x is the number of sets of which the record is a member. This value is determined by traversing the nset-node structure (see Figure 14). and comparing the record name R_i-name with nsn-member-name. For every set of which the record *is* a member. the "MEM"^S_k-name and "POS"^S_k-name attributes will be added, resulting therefore in $2x$ attributes. The final figure in the above formula, "2," refers of course to the attributes "FILE" and "DBKEY."

This concludes our review of the Template Files. We turn next to a study of the creation of the Descriptor Files.

C. CREATING THE DESCRIPTOR FILES

As has been the case with the Template File, the Descriptor File likewise must adhere to a very rigid syntax. Again, this syntax reflects the requirement of presenting MBDS with an acceptable data structure. The user participation is necessary in the creation of a Descriptor File. This stands in a marked contrast with the Template File, which has been created "automatically," with no user involvement. The Template File is used by MBDS as a general description of file components and structure; the Descriptor File, on the other hand, is used to reflect the semantic meanings and intended use of the data. In the Descriptor File, the user specifies the attributes (or fields) to be regarded as "key" or "sequence" attributes (fields). MBDS utilizes this information to create the index (cluster) arrangements that permit the most rapid and efficient response to queries and transactions when they are run against the database.

Thus, there is no single "correct" choice for a Descriptor File. The user is free to create the most efficient possible set of file descriptors from a nearly infinite variety of combinations. Figure 20 describes the basic file format. Again, the syntax reflects the attribute-based schema; the reader may translate the syntax into the relational, hierarchical, and/or network schemas as desired.

In Figure 20, DATABASE-NAME of course refers to the name of the instant database, as it has been in the Template File syntax (Figure 15). "FILE B" is a constant that will always be present; it precedes the list of file names, described next. (The meaning of the "B" in "FILE B" will be described subsequently.) Following "FILE B" is a series of lines, each beginning with an exclamation mark "!", followed by a space, then a file name. Each file (or relation, segment, or record type) is automatically included as an EQUALITY descriptor; the

```
DATABASE-NAME
FILE  B
!  Name-Of-First-Record
!  Name-Of-Second-Record
<.....>
!  Name-Of-Last-Record
@
First-Selected-Attribute-Name  A|B
{
RANGE or EQUALITY statements
}
@
Second-Selected-Attribute-Name  A|B
{
RANGE or EQUALITY statements
}
@
<.....>
Last-Selected-Attribute-Name  A|B
{
RANGE or EQUALITY statements
}
@
$
```

Figure 20. The Descriptor-File Syntax.

exclamation mark identifies the EQUALITY index term. All together then, "FILE B" followed by all the file-names in the database act to establish the basic set of descriptors that a given database will always have. Note that this sequence of descriptors ends with the at-sign, "@". This is necessary because,

unlike the Template File, there is no explicit value given to indicate the number of items to follow.

Following "Name-Of-Last-Record" and "@" is the "First-Selected-Attribute-Name A|B". What we see taking place here is the result of the presentation to the user of each attribute of each record, one-by-one. The user selects the attributes to be used as the database indexing terms. After selecting a given attribute, the user is asked whether it is to be a RANGE or an EQUALITY descriptor. The "A|B" following the selected-attribute-name reflects the user's choice of designating the attribute as a RANGE (A) *or* an EQUALITY (B) indexing term. (The vertical bar, "|", is the BNF syntax symbol for "or.") Either the *A* or the *B* will therefore follow the attribute name, and the constant "FILE B" thus represents a (mandatory) EQUALITY indexing term.

Next in the Figure is the notation

{

RANGE or EQUALITY statements

}

This is purely formalistic; neither the curly brackets nor these words appear as shown in this position. Instead, a sequence of statements reflecting the selected RANGE or EQUALITY values appear here, followed by the "@." As an example, suppose the current attribute name is CITY, and we wished to establish EQUALITY index terms for selection to include Monterey, Portland, Houston, Boston, and Seattle. This may be accomplished by the following sequence:

```
CITY  B
!  Monterey
!  Portland
!  Houston
!  Boston
!  Seattle
@
```

Alternatively, suppose we wanted to establish the attribute POPULATION as an indexing term, utilizing specified RANGE statements. The ranges that we might establish are 0 to 10000, 10001 to 50000, 50001 to 100000, and 100001 to 1000000000. The following sequence of statements fulfills our purpose:

```
POPULATION  A
0  10000
10001  50000
50001  100000
100001  1000000000
@
```

Finally, following the entry of the last selected attribute and its associated RANGE or EQUALITY statements, we note the dollar sign, "$." This acts as our file termination symbol, and--as is the case with the at-sign, "@"--is required because there is no other convenient *a priori* means to identify the file extent. The specific choices for indexing terms are ad hoc, based solely on the user's experience and desires in the establishment of the database.

At this point, it might prove useful to provide a concrete example of a Descriptor File, as we have done with the Template File. Figure 21 therefore reflects one possible set of choices in establishing a Descriptor File for the relational database of Figure 5.

Before proceeding with the algorithms themselves, it may also prove useful to describe the prompting sequence that is used to designate the desired attributes (fields), and establish them as RANGE or EQUALITY descriptors, together with

```
SCHOOL-DAYS
FILE B
! COURSE
! PREREQ
! OFFERING
! SCHEDULE
@
COURSE# B
! Cs4100
! Cs4200
! Is3100
@
DATE A
850101 850630
850901 851231
860101 860320
@
LOCATION B
! Monterey
! Carmel
! Portland
! Butte
@
$
```

Figure 21. A Relational Database Descriptor File.

their respective values. The first step is to inform the user of the general procedure. (In this case, since we are reflecting the actions of the language interface, we use the relational terminology.)

The following are the Relations in the <DatabaseName> Database:

RelName(1)
RelName(2)
<.....>
RelName(n)

Beginning with the first Relation, we will present each
Attribute of the relation. You will be prompted as to whether
you wish to include that Attribute as an Indexing Attribute,
and, if so, whether it is to be indexed based on strict
EQUALITY, or based on a RANGE OF VALUES.

Strike RETURN when ready to continue.
Action --- >

When the user has had a chance to read this message and strike the carriage return, the system begins to cycle through each attribute of each relation. The user is asked whether the attribute is to be chosen as an indexing term, and if so, of what kind (RANGE or EQUALITY). For example, imagine that the system is prompting the user by way of the COURSE# attribute of the COURSE relation:

Relation name: COURSE
Attribute name: COURSE#

Do you wish to install this Attribute as an Indexing Attribute?

(n) - no: continue with next Attribute/Relation
(e) - yes: establish this as an EQUALITY Attribute
(r) - yes; establish this as a RANGE Attribute

Action --- >

If the user selects the choice "e," the system responds with

Enter EQUALITY match value, or <CR> to exit:

The user may therefore enter an EQUALITY value, or change his mind entirely

and exit. If a value is entered, the immediately preceding prompt is repeated indefinitely, allowing the user to enter as many EQUALITY values for the instant attribute as desired. A carriage return terminates the sequence; at this point, the next attribute will be displayed, with the same three action choices.

If the "r" choice is selected, the system responds:

Enter Lower Bound, or <CR> to exit:

Again, the user may exit without prejudice with a carriage return, if he decides it has been an error to select this attribute. If he *does* enter a lower bound, the system responds next with:

Enter Upper Bound:

After responding, the system returns to the "Lower Bound" prompt, and so the session continues until the user responds to the "Lower Bound" prompt with a carriage return. At this point, the system proceeds with the next attribute.

Often an attribute is present in two or more relations. In that case, it may happen that the attribute has been identified in an earlier relation to act as an indexing attribute. When this occurs, the system notifies the user that the present attribute has been previously selected by providing the previously chosen index value(s), and asking if more are to be added. For example, suppose that we had entered "Cs2100" as an EQUALITY term for COURSE#, above. Then, in the PREREQ relation we encounter this attribute again:

Relation name: PREREQ
Attribute name: COURSE#

Cs2100

Do you wish to add more EQUALITY values? (y or n)
Action --- >

Logic exists in the system to determine whether the previously-entered descriptor term is an EQUALITY or RANGE term, and to adjust the prompt accordingly. If we respond "y," the system in turn responds with the expected

Enter EQUALITY match value. or <CR> to exit:

Again, we continue to receive this prompt until we respond with a carriage return. An analogous situation occurs for a previously selected RANGE attribute.

After all attributes of all relations have been considered in turn, the system returns control to the superordinate routine, and our task is finished.

We now move on, considering the various Descriptor-File-creation algorithms in turn, beginning with the relational algorithm.

1. The Relational Algorithm

As we note in the following sequence of algorithms, there is a sense of sameness among them: one is virtually identical to the other. At this high level of abstraction, this is perhaps not too surprising. In what follows, we endeavor to identify some of the underlying (implementation) distinctions.

The relational algorithm is shown in Figure 22. Items enclosed in double quotes, such as "FILE B" and "!" are to be written to the file as literals, i.e., precisely as specified in the algorithm. As before, the symbol "|" is interpreted to be the "or" symbol: thus, the line

write A_{R_i} A|B

means to write a line to the file in which the attribute name is followed by a space, and then by the character "A" *or* "B." The lines

write additional indexing values

and

write indexing values

are shorthand for the interactive querying process described in the preceding section, in which the EQUALITY and RANGE indexing values are derived.

Appendix D is the C-procedure that implements the algorithm of Figure 22. Again, the comparatively simple and straightforward layout of the relational data structures (Figure 12) contributes greatly to the relative simplicity of the algorithmic implementation. As shown in our algorithm, two references to the relational data structures are made: the first writes the respective relations to the file as EQUALITY index terms, while the second is used to interactively present the attributes to the user for the purpose of creating the remaining index terms.

The algorithm asks the question

if (A_{R_i} already selected as index term) then...

which we are able to deal with by creating dynamic data structures as we proceed. Each index term is thus stored in this dynamic data structure as it is initially designated by the user. Then, at each succeeding attribute, we first search through this data structure to determine whether or not the instant attribute has been selected as an index term. If it has, we are able to display the

Assertions:

1. Relational Database D has relations $\{R_1, R_2, \ldots, R_n\}$.
2. Each relation R_i, i = 1.....n. has the relation name R_i-name.
3. Each relation R_i, i = 1.....n. has A_{R_i} attributes.
4. Each attribute A_j, j = 1,...,A_{R_i}, has attribute name A_j-name.

Algorithm:

```
write D-Name        /* Database Name */
write "FILE B"
/* Repeat for each relation in database */
for each relation R_i in database D do
     {
     write "!" R_i-name
     }
write "@"
/* Repeat for each relation in database */
for each relation R_i in database D do
     {
     /* Repeat for each attribute in relation */
     for each attribute A_R_i in relation R_i do
          {
          if (A_R_i already selected as index term) then
               if (more values to be added) then
                    write additional indexing values
          else
               if (A_R_i is to be index term) then
                    write A_R_i  A|B
                    write indexing values
                    write "@"
          }
     }
write "$"
```

Figure 22. The Relational-Descriptor-File-Transformation Algorithm.

indexing values that have been previously entered, and ask the user if he wishes to enter additional values. If the attribute has *not* previously been selected, that option is then offered to the user.

In actuality, then, the various index terms are *not* written to the file at precisely the times indicated by the algorithm. Instead, the procedure in Appendix D performs this task *after* all attributes have been reviewed and all index terms have been designated by the user. Then, our dynamic data structure is traversed, and the index terms and RANGE/EQUALITY values are written to the file.

2. The Hierarchical Algorithm

As suggested earlier, the hierarchical algorithm--apart from some model-specific terminology--is essentially the same as that of the relational. Figure 23 is the algorithm for transforming the hierarchical data structures (Figure 13) into a Descriptor File.

As in the relational case, two accesses are made to the hierarchical data structure. The first writes the segment names to the file as EQUALITY indexing terms; the remainder of the indexing terms are interactively designated by the user on the second pass. As has been the case in emplate File creation, a pre-order recursive traversal technique is used to work through all segments of the hierarchy. Appendix E is the C-procedure that implements the algorithm of Figure 23.

3. The Network Algorithm

Finally, the network algorithm is presented in Figure 24; by now, this algorithm should be familiar. Only the specific implementation techniques

Assertions:

1. Hierarchical Database D has segments $\{S_1, S_2,....,S_n\}$.
2. Each segment S_i, i = 1,....n, has segment name S_i-name.
3. Each segment S_i, i = 1,....n, has F_{S_i} fields.
4. Each field F_j, j = 1,...,F_{S_i}, has field name F_j-name.

Algorithm:

```
write D-name            /* Database Name */
write "FILE  B"
/* Repeat for each segment in the database */
for each segment S_i in database D do
     {
     write  "!"  S_i-name
     }
write  "@"
/* Repeat for each segment in the database */
for each segment S_i in database D do
     {
     /* Repeat for each field in the segment */
     for each field F_S_i do
          {
          if (F_S_i already selected as index term) then
               if (more values to be added) then
                    write additional indexing values
          else
               if (F_S_i is to be index term) then
                    write F_S_i   A|B
                    write indexing values
                    write  "@"
          }
     }
write  "$"
```

Figure 23. The Hierarchical-Descriptor-File-Transformation Algorithm.

Assertions:

1. Network Database D has records $\{R_1, R_2, \ldots, R_n\}$.
2. Each record R_i, i = 1.....n, has the record name R_i-name.
3. Each record R_i, i = 1,....n, has A_{R_i} attributes.

Algorithm:

```
write D-name            /* Database Name */
write "FILE  B"
/* Repeat for each record in the database */
for each record R_i in database D do
     {
     write  "!"  R_i-name
     }
write  "@"
/* Repeat for each record in the database */
for each record R_i in database D do
     {
     /* Repeat for each attribute in the record */
     for each attribute A_R_i do
          {
          if (A_R_i already selected as index term) then
               if (more values to be added) then
                    write additional indexing values
          else
               if (A_R_i is to be index term) then
                    write A_R_i   A|B
                    write indexing values
                    write  "@"
          }
     }
write  "$"
```

Figure 24. The Network-Descriptor-File-Transformation Algorithm.

required to traverse the network data structures (Figure 14) vary from the relational or the hierarchical cases.

Appendix F is the C-procedure written to implement the algorithm of Figure 24. It should present no particular surprises by this time to the reader, as it possesses a streamlining similarity to the other two cases. As has been required in the network-Template-File creation, a recursive technique is employed to traverse the record attributes, which have a kind of hierarchical structure to them.

In the next chapter, we discuss some of the issues and results of integrating these six procedures into the multi-lingual database system and the multi-backend database system.

V. THE SYSTEM INTEGRATION

A. SOME GENERAL COMMENTS

In referring to the field of Software Engineering, this thesis deals with the issue of software system integration. Thus, the research and implementation efforts of the present work have been to effect an *integration* of two separate software systems, namely the multi-lingual database system (MLDS) and the multi-backend database system (MBDS). Each is the result of the hard work of many generations of graduate students, first at the Ohio State University, and more recently, at the Naval Postgraduate School. Furthermore, this ongoing developmental effort has been both guided and focused by the two men who, it can reasonably be stated, have the "broader picture" of the MLDS and MBDS efforts: Professor David K. Hsiao and Mr. Steven A. Demurjian.

As Frederick Brooks has noted,

> Men and months are interchangeable commodities only when a task can be partitioned among many workers *with no communication among them*....This is true of reaping wheat or picking cotton; it is not even approximately true of systems programming. [Ref. 11: p. 16]

MLDS and MBDS *are* products of "many workers," and therefore require a great deal of communications among these workers, together with coordination efforts.

An interesting observation, however, is that the communication has had more of an *inter-generation* than an *intra-generation* flavor. In other words, the

research and development efforts of this thesis have been far more affected by the work of previous thesis students. than by the concurrent work of others.

As a result, "communication" efforts have primarily focused on two resources:

1. Published theses, and
2. Thesis Advisor and Second Reader.

The work of this thesis has been largely independent of the ongoing efforts of others. This does not alter Brooks' observation; it merely reflects a form of communication that is inherently different in *nature*, but not in scope or importance. In projects as large and involved as the MLDS and MBDS endeavors, we can state with absolute assurance that the time and effort involved in training and indoctrination have more than substantiated Brooks' arguments concerning large software developmental undertakings.

A final note is made here concerning the operating system and the programming medium used in this thesis. It has been necessary to gain a working knowledge of both the Unix operating system and the C programming language, in order to competently deal with the implementation requirements of this thesis. The C programming language is a natural choice for the work performed under the Unix operating system. We have been fortunate in having a second reader who is an expert in C, as well as in Unix.

B. SPECIFIC INTEGRATION TASKS

We speak of the integration process of this thesis as two distinct phases. The first phase involves intra-MLDS integration; the second phase is concerned with the integration of MLDS and MBDS into a single, coherent entity. It is in the

latter phase that the aims of the present thesis--together with many of the theses that have preceded it--are achieved.

1. The Intra-MLDS Integration

The intra-MLDS integration effort is briefly described herein. In any large software project, the whole system consists of a large number of related program modules. Inevitably, whether by design or otherwise, some modules are completed before others; at the same time, however, a module may call or depend on other, possibly unfinished, modules. MLDS, although deemed complete, has called upon various procedures which do not exist. Thus, the common programming technique of "stubs" has been used, i.e., "dummy" call sequences and procedures have been written in order to permit the compilation and execution of these earlier procedures and programs.

With the advent of the respective procedures of the present thesis, the required integration becomes quite clear. It has been necessary to remove the procedure stubs and to adjust the calling sequences in order to properly reference the newly-written procedures. As we have noted in the preceding chapters, the creation of the Template File remains entirely transparent to the user, but the sequence to create the Descriptor File is now visible.

2. The MLDS--MBDS Integration

As one might expect, to effectively carry out the MLDS--MBDS integration, a re-working similar to that for the Intra-MLDS integration is required. MLDS becomes a component of the larger whole--the MLDS--MBDS system. Therefore, instead of a separate subsystem, certain code modifications are required to transform MLDS into a C function call. Again, the "stub" technique

has been used, and these stubs must be removed and replaced by the code for each of the relational, hierarchical, and network elements performing the corresponding interface tasks.

For example, the main system routine of MLDS--MBDS queries the user at the top level as to which function is desired:

What operation would you like to perform?

(g) - generate database
(l) - load database
(e) - execute test interface
(s) - execute the SQL interface
(d) - execute the DL/I interface
(c) - execute the CODASYL interface
(a) - execute the DAPLEX interface
(x) - exit to the operating system
(z) - exit and stop MBDS

(Note that SQL refers to the relational model, DL/I to the hierarchical model, CODASYL to the network model, and DAPLEX to the entity-relationship model.) The options functionally available to the user prior to the completion of the present work included (g) - generate database, (l) - load database, (e) - execute test interface, (x) - exit to the operating system, and (z) - exit and stop MBDS. In this thesis, the code has been generated (Appendices A - F) to permit the addition of three interfaces: (s) - execute the SQL interface, (d) - execute the DL/I interface, and (c) - execute the CODASYL interface. The DAPLEX interface is not a part of this thesis.

Among other required code modifications, it has been necessary to make certain global variables accessible to MLDS. Following these changes, the relevant portions of the system require re-compilation and re-linking; the result of these actions is an executable main system module with three language interfaces operational.

In Chapter VI. we summarize the results and conclusions of this Multi-lingual-Database-System--Multi-Backend-Database-System Interface.

VI. THE CONCLUSION

In this thesis, we have observed many of the difficulties that are associated with the conventional, single-data-model approach to the database management system (DBMS). We noted that each of the models (e.g., the relational, hierarchical, and network models) tends to have its own peculiar strengths and weaknesses, and that no single model can reasonably be expected to perform optimally in every application. The costs of installing two or more different and separate model-based systems are high. Further, performance upgrades of a conventional DBMS are difficult, and the expenses of upgrading general single-model-based systems are much higher. These problems have prompted the call for an alternative means of solving the growing database problems.

The multi-lingual database system (MLDS) and the multi-backend database system (MBDS) are offered, together, as a very attractive solution to many of the difficulties associated with the conventional DBMS. As we have seen, MLDS offers the user a choice of data models and languages. Therefore, any previously-developed applications from one of the (supported) models need not be redeveloped when they are moved to the MLDS environment. At the same time, a variety of different models are available and supported by MLDS, offering the user the opportunity to exploit their features and advantages.

MBDS, meanwhile, approaches the performance problems of conventional DBMS packages by offloading the DBMS functions onto a number of identical, parallel processors and stores. Experiments have validated the promise of this approach, identifying a *reciprocal reduction* in response times to user transactions,

as the number of backends is increased (while the database size, and therefore the size of transaction responses, are being held constant). We likewise observe a *response-time invariance* in response times, when the increase in transaction responses is matched by a proportional increase in the number of backends. Thus, stable response times are obtained.

A significant appeal of the MLDS/MBDS solution is the fact that it is *software-oriented,* using off-the-shelf hardware. In this regard, it differs radically from hardware approaches such as DBC [Ref. 12], and therefore is potentially much more attainable in the near term.

As we have seen, the operational methodology being utilized to achieve multi-lingual *and* multi-backend capabilities is the transformation. Databases, queries, and transactions are *transformed* into a single, coherent model (i.e., the attribute-based data model) and language (i.e., the attribute-based language). Clearly, this transformation, and indeed the MLDS/MBDS solution, is valid only if the overhead generated by the transformation processes does not become excessive. Experimental results to date have shown the promise of the solution, and the validity of the transformation.

In this thesis, we have implemented two facets of the interface of MLDS and MBDS. These have been on database *creation* requirements. **Template Files** have been developed for each of the relational, hierarchical, and network schemas. The Template File, produced in the language interface, is employed to describe the basic *structure* of a newly-created database to MBDS. This is important because, as we have seen, MBDS (unlike MLDS) can only "understand" one model, and can only "speak" one language: the attribute-based data model and data language, respectively. The creation of this file is automatic, requiring no user involvement.

The generation of the **Descriptor Files**, on the other hand, is largely dependent on the interaction of a knowledgeable, informed user. The Descriptor File is developed to provide an efficient indexing scheme for the storage of--and subsequent accesses from--the new database. It is therefore essential that the user, in the interactive process of developing a Descriptor File, has a solid understanding of the semantic interrelationships of the database, together with a knowledge of the probable types and patterns of the database usage. The overall efficiency of the subsequent applications is critically dependent on this phase.

As noted in Chapter V, the success of the MLDS and MBDS efforts is the result of the work of many minds and many hands. The eventual success of this, or any other large software project, is ultimately a function of the success with which the substantial software engineering challenges are met, and mastered.

```
/* This file is:      lilcommon.c                                    */
/* LAST DATE MODIFIED:  2/21/86 Created this file                    */


#include <stdio.h>
#include "licommdata.def"
#include "flags.def"
#include "sql.ext"
#include "lil.ext"



build-ddl-files()

  {
  /* This routine is used to create the MBDS template and descriptor files,  */
  /* calling on a separate routine to create each file:                       */

  struct   ddl-info   *ddl-info-alloc();

  if (sql-info-ptr -> si-ddl-files == NULL)
     sql-info-ptr -> si-ddl-files = ddl-info-alloc();

  build-template-file();

  build-desc-file();

  }  /*  End "build-ddl-files()" routine  */
```

```
build-template-file()

  {
   * This routine builds the MBDS template file for a new Relational      *
   * Database that was just created:                                       */

  struct rel-dbid-node   *db-ptr;      /* Database pointer                 *
  struct rel-node        *rel-ptr;     /* Pointer to a Relational node     */
  struct rattr-node      *at-ptr;      /* Pointer to a Relational Attribute */
  struct file-info       *f-ptr;       /* File pointer                     */
  char                   temp-str[NUMDIGIT + 1]; /* Temporary string       */

  /* Begin by setting the pointers to the sql-info data structure         */
  /* that is maintained for each user of the system:                      *
  db-ptr = sql-info-ptr->si-curr-db.cdi-db.dn-rel;
  f-ptr  = &(sql-info-ptr->si-ddl-files->ddli-temp):

  /* Next, copy the filename where the MBDS template information will     */
  /* be stored.  This filename is constant and was obtained from          *
  /* licommdata.def:                                                      */
  strcpy(f-ptr->fi-fname, RTEMPFname);

  /*  Next, open the template File to be created, for Write access:       *
  f-ptr->fi-fid = fopen(f-ptr->fi-fname,"w");

  /*  Next. write out the database name & number of relations:            */
  fprintf(f-ptr->fi-fid, "%s\n", db-ptr->rdn-name);
  num-to-str(db-ptr->rdn-num-rel, temp-str);
  fprintf(f-ptr->fi-fid, "%s\n", temp-str);

  /*  Next, set the pointer to the first relation:                        */
  rel-ptr = db-ptr->rdn-first-rel;

  /*  While there are more relations to process, write out the number     *
  /*  of attributes (+1 for the attribute "FILE"), and the relation name: */
  while (rel-ptr != NULL)
    {
    num-to-str((rel-ptr->rn-num-attr + 1), temp-str);
    fprintf(f-ptr->fi-fid, "%s\n", temp-str);
    fprintf(f-ptr->fi-fid, "%s\n", rel-ptr->rn-name);

    /*  Now. set the pointer to the first attribute:                       *
    at-ptr = rel-ptr->rn-first-attr;

    /*  Next, print out the constant attribute "FILE s", and while there  */
    /*  are more attributes to process, print out each attribute name      *
    /*  and type:                                                          *
    fprintf(f-ptr->fi-fid, "FILE s\n");
    while (at-ptr != NULL)
      {
      fprintf(f-ptr->fi-fid, "%s %c\n", at-ptr->ran-name.
```

```
                at-ptr->ran-type.f-ptr->fi-fid);

        * Set the pointer to the next attribute:                    *
      at-ptr = at-ptr->ran-next-attr;
      } /* End "while (at-ptr != NULL)" */

    /* set the pointer to the next relation:                        */
    rel-ptr = rel-ptr->rn-next-rel;
    } /* End "while (rel-ptr != NULL)" */

  /* Finally, close out the file and exit this routine:            */
  fclose(f-ptr->fi-fid);

} /* End "build-template-file()" routine */
```

```
#include <stdio.h>
#include "licommdata.def"
#include "flags.def"
#include "dli.ext"
#include "lil.ext"


build-ddl-files()

  {
  /* This routine is used to create the MBDS template and descriptor files,  */
  /* calling a separate routine for the creation of each:                     */

  struct   ddl-info   *ddl-info-alloc();

  if (dli-info-ptr -> di-ddl-files == NULL)
     dli-info-ptr -> di-ddl-files = ddl-info-alloc();

  build-hie-template-file();

  build-desc-file();

  }  /*  End "build-ddl-files()" routine  */
```

```
build-hie-template-file()
  {

  /* This routine builds the MBDS template file for a new hierarchical     */
  /* database that was just created:                                       */

  struct  hie-dbid-node   *db-ptr;     /* ptr to root node in hierarchy    */
  struct  hrec-node       *hie-ptr;    /* ptr to an hierarchical node      */
  char                    temp-str[NUMDIGIT + 1];  /* a "temporary string" */
  struct  file-info       *f-ptr;      /* file pointer                     */

  /* Begin by setting the pointers to the dli-info data structure that is  */
  /* maintained for each user of the system:                               */
  db-ptr = dli-info-ptr->di-curr-db.cdi-db.dn-hie;
  f-ptr = &(dli-info-ptr->di-ddl-files->ddli-temp);

  /* Next, copy the filename where the MBDS template information will      */
  /* be stored.  This filename is a Constant, and was obtained from        */
  /* "licommdata.def":                                                     */
  strcpy(f-ptr->fi-fname, HTEMPFname);

  /* Now, open the template file to be created, for Write access:          */
  f-ptr->fi-fid = fopen(f-ptr->fi-fname,"w");

  /* Next, write out the database name and the number of segments:         */
  fprintf(f-ptr->fi-fid, "%s\n",db-ptr->hdn-name);
  num-to-str(db-ptr->hdn-num-seg, temp-str);
  fprintf(f-ptr->fi-fid, "%s\n",temp-str);

  /* Now, set the database pointer to the root segment, and call a         */
  /* routine to traverse the hierarchical structure, writing out the       */
  /* appropriate segment and attribute information:                        */
  hie-ptr = db-ptr->hdn-root-seg;
  hie-traverse(hie-ptr);

  /* Finally, close the file and end the program:                          */
  fclose(f-ptr);

  } /* End "build-hie-template-file()" routine */
```

```
hie-traverse(hie-ptr)

  struct  hrec-node   *hie-ptr;   /*  Pointer to an hierarchical node    *

  {

   *  This routine performs a pre-order recursive traversal of an          */
  /*  hierarchical data structure:                                         */

  visit-hie-node(hie-ptr);

  if (hie-ptr->hn-first-child != NULL)
    hie-traverse(hie-ptr->hn-first-child);

  if (hie-ptr->hn-next-sib != NULL)
    hie-traverse(hie-ptr->hn-next-sib);

  }  /*  End "hie-traverse(...)" routine  */
```

```
visit-hie-node(hie-ptr)

  struct  hrec-node   *hie-ptr;     /*  Pointer to an hierarchical node     */

  {
  /*  This routine performs the required actions on each node in the       */
  /*  hierarchical structure; this involves tracing a route from the       */
  /*  "current" node back to the root, and keeping track of all key attri-  */
  /*  butes from all nodes along that route.  Then, these attributes,       */
  /*  together with all attributes from the "current" node, will be written */
  /*  to the Template File:                                                 */

  struct  temp-node  *temp-node-alloc(),      /* Allocates Temp. Nodes    */
                 *head-ptr,           /* Pointers to Nodes Used.. */
                 *tempnode-ptr;         /* ...by this Routine       */
  struct  hattr-node  *hattr-ptr;          /*  Attribute Pointer       */
  struct  hrec-node   *parent-ptr;         /*  Pointer to Node's Parent */
  int               attr-ctr = 0;         /*  Attribute Counter       */
  char              temp-str[NUMDIGIT + 1]; /*  A "Temporary String"    */
  FILE              *file-ptr;            /*  File Pointer            */

  /*  First, we initialize the file pointer:                            */
  file-ptr = dli-info-ptr->di-ddl-files->ddli-temp.fi-fid;

  /*  Second, initialize the "head-ptr" (which will point at the linked list */
  /*  that we are about to create to hold the key attributes from each node  */
  /*  along the path back to the root).  Then, begin to trace the path       */
  /*  back to the root:                                                 */
  head-ptr = NULL;
  parent-ptr = hie-ptr->hn-parent;
  while (parent-ptr != NULL)
    {

    /*  Begin with the first attribute, and then check all attributes in    */
    /*  this node to see if they are "key" attributes, to be saved in the   */
    /*  linked list:                                                        */
    hattr-ptr = parent-ptr->hn-first-attr;
    while (hattr-ptr != NULL)
      {
      if (hattr-ptr->han-key-flag == TRUE)
        {

        /*  This IS a "key" attribute, so we want to save the attribute     */
        /*  name & type in our linked list structure.  Also, "bump" the     */
        /*  attribute counter to record the effective number of attri-      */
        /*  butes to be recorded for this segment:                          */
        attr-ctr++;
        tempnode-ptr = temp-node-alloc();
        strcpy(tempnode-ptr->tn-attr-name, hattr-ptr->han-name);
        tempnode-ptr->tn-type = hattr-ptr->han-type;
        tempnode-ptr->tn-next-node = head-ptr;
```

```
        head-ptr = tempnode-ptr;
        tempnode-ptr = NULL;
        } /* End "if (hattr-ptr->han-key-flag == TRUE)" */

      /* Continue with the next attribute in this node:          */
      hattr-ptr = hattr-ptr->han-next-attr;
      } /* End "while (hattr-ptr != NULL)" */

    /* Continue with the next node along the path to the root:    */
    parent-ptr = parent-ptr->hn-parent;
    } /* End "while (parent-ptr != NULL)" */

/* Having finished constructing our linked list, we may continue to pro- */
/* cess the "current" node. First, calculate the effective number of     */
/* attributes, which is the number recorded in our linked list along the */
/* path to the root, plus the number in the "current" node itself, plus  */
/* one (for the constant attribute "FILE s"); then, write this value,    */
/* together with the segment name, to the file:                          */
num-to-str((hie-ptr->hn-num-attr - attr-ctr + 1), temp-str);
fprintf(file-ptr, "%s\n", temp-str);
fprintf(file-ptr, "%s\n", hie-ptr->hn-name);

/* Now, we can print the constant attribute "FILE s", and then continue  */
/* by traversing our linked list, printing out the attribute names and   */
/* types from the "key" attributes located along the path to the root:   */
fprintf(file-ptr, "FILE s\n");
tempnode-ptr = head-ptr;
while (tempnode-ptr != NULL)
  {
  fprintf(file-ptr, "%s %c\n",tempnode-ptr->tn-attr-name.
        tempnode-ptr->tn-type);
  tempnode-ptr = tempnode-ptr->tn-next-node;
  } /* End "while (tempnode-ptr != NULL)" */

/* At last, we can process the "current" node itself, traversing through */
/* the node's attributes, printing their respective names & types:       */
hattr-ptr = hie-ptr->hn-first-attr;
while (hattr-ptr != NULL)
  {
  fprintf(file-ptr, "%s %c\n",hattr-ptr->han-name,hattr-ptr->han-type);
  hattr-ptr = hattr-ptr->han-next-attr;
  } /* End "while (hattr-ptr != NULL)" */

/* Finally, we want to free the previously allocated linked list memory: */
while (head-ptr != NULL)
  {
  tempnode-ptr = head-ptr->tn-next-node;
  free(head-ptr);
  head-ptr = tempnode-ptr;
  } /* End "while (head-ptr != NULL)" */
} /* End "visit-hie-node(hie-ptr)" routine */
```

APPENDIX C - THE NETWORK TEMPLATE FILE

```
#include <stdio.h>
#include "licommdata.def"
#include "flags.def"
#include "dml.ext"
#include "lil.ext"

build-ddl-files()
/*  This routine is used to create the MBDS template and descriptor files,    */
/*  calling a separate routine to create each file:                            */

  {
  struct  ddl-info    *ddl-info-alloc();

  if (dml-info-ptr->dmi-ddl-files == NULL)
     dml-info-ptr->dmi-ddl-files = ddl-info-alloc();

  build-net-template-file();

  build-desc-file();

  }  /*  End "build-ddl-files()" routine  */
```

```
build-net-template-file()

{
/*  This routine builds the MBDS template file for a new network database  */
/*  that was just created:                                                 */

struct  file-info       *f-ptr;       /* File Pointer                     */
struct  net-dbid-node   *db-ptr;      /* Database Pointer                 */
struct  nrec-node       *net-ptr;     /* Network Node (record) Pointer    */
struct  nset-node       *nset-ptr;    /* Network Set Pointer              */
struct  nattr-node      *at-ptr;      /* Network Attribute Pointer        */
int                  member-ctr;   /* # of Sets Record is a Mbr of     */
char                 temp-str[NUMDIGIT + 1]; /* A Temporary String     */

/*  Begin by setting the pointers to the dml-info data structure        */
/*  that is maintained for each user of the system:                     */
db-ptr = dml-info-ptr->dmi-curr-db.cdi-db.dn-net;
f-ptr = &(dml-info-ptr->dmi-ddl-files->ddli-temp);

/*  Next, copy the filename where the MBDS template information will    */
/*  be stored.  This filename is a Constant, and was obtained from      */
/*  licommdata.def:                                                     */
strcpy(f-ptr->fi-fname, NTEMPFname);

/*  Now, open the template file to be created, for Write Access:        */
f-ptr->fi-fid = fopen(f-ptr->fi-fname, "w");

/*  Next, write out the database name and the number of records:        */
fprintf(f-ptr->fi-fid, "%s\n", db-ptr->ndn-name);
num-to-str(db-ptr->ndn-num-rec, temp-str);
fprintf(f-ptr->fi-fid, "%s\n", temp-str);

/*  Now, set the database pointer to the first record:                  */
net-ptr = db-ptr->ndn-first-rec;

/*  While there are more records to process, traverse the Linked List of  */
/*  records, writing out the number of attributes for each record.  This  */
/*  number is obtained by summing the nrec-node field "nrn-num-attr",     */
/*  One for FILE, One for DBKEY, and One for each set membership, MEM,     */
/*  and one for the relative position within that set, POS:              */
while (net-ptr != NULL)

  /*  For each record, traverse the "nset-node" linked list to determine  */
  /*  whether the record is a MEMBER of any sets:                         */
  {
  member-ctr = 0;
  nset-ptr = db-ptr->ndn-first-set;
  while (nset-ptr != NULL)
    {
    if (strcmp(net-ptr->nrn-name, nset-ptr->nsn-member-name) == 0)
      member-ctr++;
```

```
    nset-ptr = nset-ptr->nsn-next-set;
    } /* End "while (nset-ptr != NULL)" */

  /* Now, calculate the effective number of attributes for this record. */
  /* print this value out, together with the record name, and the con- */
  /* stant attributes FILE and DBKEY:                                  */
  num-to-str((net-ptr->nrn-num-attr + (2 * member-ctr) - 2), temp-str);
  fprintf(f-ptr->fi-fid, "%s\n", temp-str);
  fprintf(f-ptr->fi-fid, "%s\n", net-ptr->nrn-name);
  fprintf(f-ptr->fi-fid, "FILE s\n");
  fprintf(f-ptr->fi-fid, "DBKEY i\n");

  at-ptr = net-ptr->nrn-first-attr;
  nattr-traverse(at-ptr, f-ptr);

  /* If the current record IS a Member of any set (determined by check- */
  /* ing the value of "member-ctr", which was incremented once in the   */
  /* previous traversal of the linked list for every set that the record */
  /* is a Member of), we must write the appropriate MEMber and POSition */
  /* information to the file. Therefore, run through the linked list    */
  /* once more, to concatenate the proper names to write to the file:   */
  if (member-ctr != 0)
    {
    nset-ptr = db-ptr->ndn-first-set;
    while (nset-ptr != NULL)
      {
      if (strcmp(net-ptr->nrn-name, nset-ptr->nsn-member-name) == 0)
        {
        fprintf(f-ptr->fi-fid, "MEM");
        fprintf(f-ptr->fi-fid, "%s i\n", nset-ptr->nsn-name);
        fprintf(f-ptr->fi-fid, "POS");
        fprintf(f-ptr->fi-fid, "%s i\n", nset-ptr->nsn-name);
        } /* End "if (strcmp(...,...) == 0)" */
      nset-ptr = nset-ptr->nsn-next-set;
      } /* End "while (nset-ptr != NULL)" */

    } /* End "if (member-ctr != 0)" */

  net-ptr = net-ptr->nrn-next-rec;
  } /* End "while (net-ptr != NULL)" */

} /* End "build-net-template-file()" routine */
```

```
nattr-traverse(at-ptr, f-ptr)

  struct   nattr-node   *at-ptr;   /* Network Attribute Pointer          */
  struct   file-info    *f-ptr;    /* File Pointer                       */

  {
  fprintf(f-ptr->fi-fid, "%s %c\n", at-ptr->nan-name, at-ptr->nan-type);

  if (at-ptr->nan-child != NULL)
    nattr-traverse(at-ptr->nan-child, f-ptr);
  if (at-ptr->nan-next-attr != NULL)
    nattr-traverse(at-ptr->nan-next-attr, f-ptr);
  }  /*  End "nattr-traverse(...)" routine  */
```

```
#include "licommdata.def"
#include "lil.ext"
#include "sql.ext"
#include <strings.h>
#include <ctype.h>

build-desc-file()

  {
  /* This routine builds the Descriptor File to be used by the MBDS in the   */
  /* creation of indexing clusters:                                          */

  struct   rel-dbid-node    *db-ptr;          /* Database Pointer      */
  struct   rel-node         *rel-ptr;         /* Relation Node Ptr     */
  struct   rattr-node       *at-ptr;          /* Attribute Node Ptr    */
  struct   descriptor-node  *desc-head-ptr,   /* Pointers to Desc-node..*/
                            *descriptornode-ptr,     /* ...Linked List  */
                            *descriptor-node-alloc(); /* Allocates Nodes */
  struct   value-node       *valuenode-ptr;   /* points to Value Node  */
  struct   file-info        *f-ptr;           /* File pointer          */
  int                       num,              /* holds User Response   */
                            found,            /* Boolean flag          */
                            goodanswer;       /* Boolean flag          */
  int                       index,            /* Loop Index            */
                            str-len;          /* Length of Relation Name*/

  /* Begin by setting the pointers to the sql-info data structure that is  */
  /* maintained for each user of the system:                               */
  db-ptr = sql-info-ptr->si-curr-db.cdi-db.dn-rel;
  f-ptr = &(sql-info-ptr->si-ddl-files->ddli-desc);

  /* Next, copy the filename where the MBDS Descriptor File information    */
  /* will be stored. This filename is Constant, and was obtained from      */
  /* licommdata.def:                                                       */
  strcpy(f-ptr->fi-fname, RDESCFname);

  /* Now, open the Descriptor File to be created, for Write access:        */
  f-ptr->fi-fid = fopen(f-ptr->fi-fname, "w");

  /* The next step is to traverse the Linked List of relations in the data- */
  /* base. There are two reasons for doing so: First, to write the Re-      */
  /* lation Names to the Descriptor File as EQUALITY Descriptors; this is   */
  /* done automatically with any Relational Database, is a necessary ele-   */
  /* ment of any Descriptor File created from such a Database, and requires */
  /* no user involvement. Second, it allows us to present the Relation      */
  /* Names (without their respective Attributes) to the User, as a memory   */
```

```
/* jog:                                                    */
system("clear");
fprintf(f-ptr->fi-fid, "%s\n", db-ptr->rdn-name):
fprintf(f-ptr->fi-fid, "FILE B\n");
printf("\nThe following are the Relations in the ");
printf("%s", db-ptr->rdn-name);
printf(" Database:\n\n");
rel-ptr = db-ptr->rdn-first-rel;

/* Traverse the Relational structure:                      */
while (rel-ptr != NULL)
  {
  fprintf(f-ptr->fi-fid, "! ");
  fprintf(f-ptr->fi-fid, "%c", rel-ptr->rn-name[0] );
  str-len = strlen( rel-ptr->rn-name );
  for(index = 1; index < str-len; index++)
    if (isupper(rel-ptr->rn-name[index]))
        fprintf(f-ptr->fi-fid, "%c", tolower( rel-ptr->rn-name[index] ));
    else
        fprintf(f-ptr->fi-fid, "%c", rel-ptr->rn-name[index]);
  fprintf(f-ptr->fi-fid, "\n");
  printf("\n\t%s", rel-ptr->rn-name);
  rel-ptr = rel-ptr->rn-next-rel;
  }  /* End "while (rel-ptr != NULL)" */

/* Each Descriptor Block must be followed by the "@" sign:  */
fprintf(f-ptr->fi-fid, "@\n");

/* Now, inform the user of the procedure that must be followed to create */
/* the Descriptor File:                                    */
printf("\n\nBeginning with the first Relation, we will present each");
printf("\nAttribute of the relation.  You will be prompted as to whether");
printf("\nyou wish to include that Attribute as an Indexing Attribute,");
printf("\nand, if so, whether it is to be indexed based on strict");
printf("\nEQUALITY, or based on a RANGE OF VALUES.");
printf("\n\nStrike RETURN when ready to continue.");
sql-info-ptr->si-answer = get-ans(&num);

/* Initialize the pointer to a Linked List that will hold the results    */
/* of the Descriptor Values, then return to the first Relation of the    */
/* database and begin cycling through the individual attributes:         */
desc-head-ptr = NULL;
rel-ptr = db-ptr->rdn-first-rel;
while (rel-ptr != NULL)
  {
  at-ptr = rel-ptr->rn-first-attr;
  while (at-ptr != NULL)
    {
    system("clear");
    printf("Relation name:  %s\n",rel-ptr->rn-name);
    printf("Attribute Name: %s\n\n",at-ptr->ran-name);
```

```
/*  Now, traverse the Attribute linked list that is being created,  */
/*  to see if the current Attribute has already been established as  */
/*  a Descriptor Attribute.  If so, offer the user the opportunity   */
/*  to add additional EQUALITY or RANGE OF VALUE values; otherwise,  */
/*  offer the user the opportunity to establish this as a Descriptor */
/*  Attribute:                                                       */
descriptornode-ptr = desc-head-ptr;
found = FALSE;
while ((descriptornode-ptr != NULL) && (found == FALSE))
  {
  if (strcmp(at-ptr->ran-name, descriptornode-ptr->attr-name) == 0)
    {

    /*  The Attribute HAS already been chosen as a Descriptor.     */
    /*  Allow the user the option of adding additional Descriptor  */
    /*  values, after listing those already entered:               */

    printf("\nThis Attribute has been chosen as ");
    printf("an Indexing Attribute.\n");
    printf("The following are the values that ");
    printf("have been specified:\n\n");
    found = TRUE;
    valuenode-ptr = descriptornode-ptr->first-value-node;
    while (valuenode-ptr != NULL)
      {
      if (descriptornode-ptr->descriptor-type == 'A')
        printf("\t%s %s\n", valuenode-ptr->value1,
                           valuenode-ptr->value2);
      else
        printf("\t%s\n", valuenode-ptr->value2);
      valuenode-ptr = valuenode-ptr->next-value-node;
      }  /*  End "while (valuenode-ptr != NULL)"  */
    printf("\nDo you wish to add more ");
    if (descriptornode-ptr->descriptor-type == 'A')
      printf("RANGE");
    else
      printf("EQUALITY");
    printf(" values?  (y or n)\n");
    sql-info-ptr->si-answer = get-ans(&num);
    if ((sql-info-ptr->si-answer == 'y') ||
       (sql-info-ptr->si-answer == 'Y'))

      /*  The user DOES wish to add more descriptors to the      */
      /*  currently existing list:                               */
      {
      if (descriptornode-ptr->descriptor-type == 'A')
        build-RAN-descrip(descriptornode-ptr, at-ptr->ran-length);
      else
        build-EQ-descrip(descriptornode-ptr, at-ptr->ran-length);
      }  /*  End "if ((sql-info-ptr->si-answer == 'y') ||
                  (sql-info-ptr->si-answer == 'Y'))        */
```

```
    }  /* End "if (strcmp(...) == 0)" */
  descriptornode-ptr = descriptornode-ptr->next-desc-node:
  }  /* End "while ((descriptornode-ptr != NULL) && (found..))" */

if (found == FALSE)

  /* The Attribute has NOT previously been chosen as a Descriptor. */
  /* Allow the user the option of making this a Descriptor Attri- */
  /* bute, with appropriate Descriptor Values:                    */
  {
  printf("\nDo you wish to install this Attribute as an ");
  printf("Indexing Attribute?\n\n");
  printf("\t(n) - no; continue with next Attribute/Relation\n");
  printf("\t(e) - yes; establish this as an EQUALITY Attribute\n");
  printf("\t(r) - yes; establish this as a RANGE Attribute\n");
  goodanswer = FALSE;
  while (goodanswer == FALSE)
    {
    sql-info-ptr->si-answer = get-ans(&num);

    switch(sql-info-ptr->si-answer)
      {
      case 'n':  /* User does NOT want to use this as an       */
                 /* Indexing (Descriptor) Attribute:           */
                 goodanswer = TRUE;
                 break;

      case 'e':  /* User wants to use this as an EQUALITY      */
                 /* Attribute:                                 */
                 goodanswer = TRUE;
                 descriptornode-ptr = descriptor-node-alloc();
                 descriptornode-ptr->next-desc-node =
                   desc-head-ptr;
                 desc-head-ptr = descriptornode-ptr;
                 strcpy(descriptornode-ptr->attr-name,
                      at-ptr->ran-name);
                 descriptornode-ptr->descriptor-type = 'B';
                 descriptornode-ptr->first-value-node = NULL;
                 build-EQ-descrip(descriptornode-ptr,
                           at-ptr->ran-length);
                 break;

      case 'r':  /* User wants to use this as a RANGE Attribute: */
                 goodanswer = TRUE;
                 descriptornode-ptr = descriptor-node-alloc();
                 descriptornode-ptr->next-desc-node =
                   desc-head-ptr;
                 desc-head-ptr = descriptornode-ptr;
                 strcpy(descriptornode-ptr->attr-name,
                      at-ptr->ran-name);
                 descriptornode-ptr->descriptor-type = 'A';
```

```
                descriptornode-ptr->first-value-node = NULL;
                build-RAN-descrip(descriptornode-ptr,
                        at-ptr->ran-length);
                break;

        default:  /* User did not select a valid choice:  */
                printf("\nError - Invalid operation ");
                printf("selected;\n");
                printf("Please pick again\n");
                break;

        } /* End Switch */

      } /* End "While (goodanswer = FALSE)" */

    } /* End "if (found == FALSE)" */

  at-ptr = at-ptr->ran-next-attr;
  } /* End "while (at-ptr != NULL)" */

rel-ptr = rel-ptr->rn-next-rel;
} /* End "while (rel-ptr != NULL)" */

/* Now, we will traverse the Linked List of Descriptor Attributes and  */
/* Values which was created, writing them to our Descriptor File:      */
descriptornode-ptr = desc-head-ptr;
while (descriptornode-ptr != NULL)
  {
  if (descriptornode-ptr->first-value-node != NULL)
    {
    fprintf(f-ptr->fi-fid, "%s %c\n", descriptornode-ptr->attr-name,
                        descriptornode-ptr->descriptor-type);
    valuenode-ptr = descriptornode-ptr->first-value-node;
    while (valuenode-ptr != NULL)
      {
      fprintf(f-ptr->fi-fid,"%s %s\n", valuenode-ptr->value1,
                        valuenode-ptr->value2);
      valuenode-ptr = valuenode-ptr->next-value-node;
      } /* End "while (valuenode-ptr != NULL)" */
    fprintf(f-ptr->fi-fid,"@\n");
    } /* End "if (descriptornode-ptr->first-value-node != NULL) */
  descriptornode-ptr = descriptornode-ptr->next-desc-node;
  } /* End "while (descriptornode-ptr != NULL)" */
fprintf(f-ptr->fi-fid, "$\n");
} /* End "build-desc-file()" routine */
```

build-EQ-descrip(descriptornode-ptr, attr-length)

```
struct   descriptor-node    *descriptornode-ptr; /* ptr to a desc.node*/
int                         attr-length;        /* length of an attr */

{
/*  This routine builds the EQUALITY Descriptor list for the current     */
/*  Field:                                                               */


struct   value-node      *valuenode-ptr,     /*  Points to Value Node  */
                        *value-node-alloc(); /*  Allocates Value Nodes */
int                      end-routine;        /*  Boolean flag          */
int                      index;              /*  Loop Index            */
int                      loop-count;         /*  Loop Index            */
int                      val-count;          /*  Loop Index            */
int                      str-len;            /*  Length of User Respon.*/
char                     *temp-value;        /*  Holds answer          */
char                     *var-str-alloc();   /*  Allocates Str.        */

/*  Repetitively offer the user the opportunity to create EQUALITY de-   */
/*  scriptors for the current Field, halting when the user enters an     */
/*  empty carriage return ("<CR>").                                      */

temp-value = var-str-alloc(attr-length + 1);
end-routine = FALSE;
while (end-routine == FALSE)
  {
  printf("\nEnter EQUALITY match value, or <CR> to exit:");
  readstr(stdin, temp-value);
  str-len = strlen( temp-value );
  for (index = 0; temp-value[index] == ' ': index++)
    ;
  if ( str-len != index )
    {
    valuenode-ptr = value-node-alloc(attr-length);
    valuenode-ptr->next-value-node = descriptornode-ptr->first-value-node;
    descriptornode-ptr->first-value-node = valuenode-ptr;
    strcpy(valuenode-ptr->value1, "!");

    /* Convert first character in temp-value to upper case if necessary: */
    if (islower(temp-value[index]))
      temp-value[index] = toupper( temp-value[index] );

    /* Convert remaining chars in temp-value to lower case if necessary: */
    for( loop-count = index - 1; loop-count <= str-len; loop-count++)
     if (isupper(temp-value[loop-count]))
       temp-value[loop-count] = tolower( temp-value[loop-count] );

    /* Store temp-value into value2 from index to end of string.        */
    val-count = 0;
```

```
    for( loop-count = index; loop-count <= str-len; loop-count--)
      valuenode-ptr->value2 val-count--  = temp-value loop-count;
    valuenode-ptr->value2 val-count  = '\0';
    } /* End "if (str-len != index)" */
  else
    end-routine = TRUE;
  } /* End "while (end-routine == FALSE)" */
free(temp-value);
} /* End "build-EQ-descrip(...)" routine */
```

```
build-RAN-descrip(descriptornode-ptr. attr-length)

struct    descriptor-node     *descriptornode-ptr: /* ptr to a desc.node*/
int                           attr-length;        /* length of an attr */

{
/*  This routine builds the RANGE OF VALUEs Descriptor list for the      */
/*  current Field:                                                       */


struct    value-node      *valuenode-ptr,     /*  Points to Value Node  */
                         *value-node-alloc(); /*  Allocates Value Nodes */
int                       end-routine;        /*  Boolean flag          */
int                       good-upper-value;   /*  Boolean flag          */
int                       index;              /*  Loop Index            */
int                       loop-count;         /*  Loop Index            */
int                       val-count;          /*  Loop Index            */
int                       str-len;            /*  Length of Input       */
char                      *temp-value;        /*  Holds Answer          */
char                      *var-str-alloc();   /*  Allocates String      */

/*  Repetitively offer the user the opportunity to create RANGE OF VALUE  */
/*  Descriptors for the current Field, halting when the user enters       */
/*  an empty carriage return ("<CR>").                                    */

temp-value = var-str-alloc(attr-length + 1);
end-routine = FALSE;
while (end-routine == FALSE)
  {
  printf("\nEnter Lower Bound. or <CR> to exit:");
  readstr(stdin, temp-value);
  str-len = strlen( temp-value );
  for (index = 0; temp-value[index] == ' '; index++)
    ;
  if ( str-len != index )
    {
    valuenode-ptr = value-node-alloc();
    valuenode-ptr->next-value-node = descriptornode-ptr->first-value-node;
    descriptornode-ptr->first-value-node = valuenode-ptr:

    /* Convert first character in temp-value to upper case if necessary: */
    if (islower(temp-value[index]))
      temp-value[index] = toupper( temp-value[index] );

    /* Convert remaining chars in temp-value to lower case if necessary: */
    for( loop-count = index + 1; loop-count <= str-len; loop-count++)
     if (isupper(temp-value[loop-count]))
       temp-value[loop-count] = tolower( temp-value[loop-count] );

    /* Store temp-value into value1 from index to end of string:       */
    val-count = 0;
```

```
    for( loop-count = index; loop-count <= str-len; loop-count--)
      valuenode-ptr->value1[val-count--] = temp-value[loop-count];
    valuenode-ptr->value1[val-count] = '\0';

    good-upper-value = FALSE;
    while (good-upper-value == FALSE)
      {
      printf("\nEnter Upper Bound:");
      readstr(stdin, temp-value);
      str-len = strlen(temp-value);
      for (index = 0; temp-value[index] == ' '; index++)
        ;
      if (str-len != index)
        {
        /* Convert first character in temp-value to upper case if nec: */
        if (islower(temp-value[index]))
          temp-value[index] = toupper( temp-value[index] );

        /* Convert remaining chars in temp-value to lower case if nec: */
        for( loop-count = index + 1; loop-count <= str-len; loop-count++)
          if (isupper(temp-value[loop-count]))
            temp-value[loop-count] = tolower( temp-value[loop-count] );

        /* Store temp-value into value1 from index to end of string:  */
        val-count = 0;
        for( loop-count = index; loop-count <= str-len; loop-count++)
          valuenode-ptr->value2[val-count++] = temp-value[loop-count];
        valuenode-ptr->value2[val-count] = '\0';

        good-upper-value = TRUE;
        } /* End "if (str-len != index)" */
      else
        printf("\nYou must supply a non-blank Upper Bound.\n");
      } /* End "while (good-upper-value == FALSE)" */
    } /* End "if (str-len != index)" */
  else
    end-routine = TRUE;
  } /* End "while (end-routine == FALSE)" */
} /* End "build-RAN-descrip(...)" routine */
```

```
#include <stdio.h>
#include <strings.h>
#include <ctype.h>
#include "licommdata.def"
#include "dli.ext"
#include "lil.ext"

struct   descriptor-node  *desc-head-ptr;   /* head-ptr to Descriptor node*/

build-desc-file()

  {
  /*  This routine builds the Descriptor File to be used by the MBDS in the  */
  /*  creation of Indexing Clusters:                                          */

  struct   hie-dbid-node   *db-ptr;          /* ptr to root node in hier.  */
  struct   hrec-node       *hie-ptr;         /* ptr to hierarchical node   */
  struct   descriptor-node *descriptornode-ptr; /* ptr to Descriptor node  */
  struct   value-node      *valuenode-ptr;   /* ptr to Value node          */
  struct   file-info       *f-ptr;           /* file pointer               */
  int                       num,             /* user query response        */
                            found,           /* Boolean flag               */
                            goodanswer,      /* Boolean flag               */
                            index;           /* loop index                 */

  /*  Begin by setting the pointers to the dli-info data structure that is   */
  /*  maintained for each user of the system:                                */
  db-ptr = dli-info-ptr->di-curr-db.cdi-db.dn-hie;
  f-ptr  = &(dli-info-ptr->di-ddl-files->ddli-desc);

  /*  Next, copy the filename where the MBDS Descriptor File information     */
  /*  will be stored.  This filename is a Constant, and was obtained from    */
  /*  licommdata.def:                                                        */
  strcpy(f-ptr->fi-fname, HDESCFname);

  /*  Now, open the Descriptor File to be created, in the Write mode:        */
  f-ptr->fi-fid = fopen(f-ptr->fi-fname, "w");

  /*  The next step is to traverse the Segments of the Hierarchical File.    */
  /*  There are two reasons for doing so:  First, to write the Segment Names */
  /*  to the Descriptor File as EQUALITY Descriptors; this is done automati- */
  /*  cally with any Hierarchical Database, forms a necessary part of the    */
  /*  Descriptor File created from such a Database, and requires no User     */
  /*  involvement.  Second, it allows us to present the Segment names        */
  /*  (without their respective Fields) to the User, as a memory jog:        */
  system("clear");
```

```
fprintf(f-ptr->fi-fid, "%s\n", db-ptr->hdn-name);
fprintf(f-ptr->fi-fid, "FILE B\n");
printf("\nThe following are the Segments in the ");
printf("%s", db-ptr->hdn-name);
printf(" Database:\n\n");

/* Call a routine to traverse the Hierarchical Database, writing the seg- */
/* ments to the File in the prescribed format, as well as printing them   */
/* to the screen.  A Linked List is also initialized here, to be used     */
/* later to hold designated Descriptor Values:                            */
hie-ptr = db-ptr->hdn-root-seg;
desc-head-ptr = NULL;
traverse-hierarchy(hie-ptr, f-ptr, FIRSTTIME);

/* Each Descriptor Block must be followed by the "@" sign:                */
fprintf(f-ptr->fi-fid, "@\n");

/* Now, inform the user of the procedure that must be followed to create  */
/* the Descriptor File:                                                   */
printf("\n\nBeginning with the first Segment, we will present each");
printf("\nField of that Segment.  You will be prompted as to whether");
printf("\nyou wish to include that Field as an Indexing Field,");
printf("\nand, if so, whether it is to be indexed based on strict");
printf("\nEQUALITY, or based on a RANGE OF VALUES.");
printf("\n\nStrike RETURN when ready to continue.");
dli-info-ptr->di-answer = get-ans(&num);

/* Re-set the Hierarchical Segment pointer to the Root Segment, then call */
/* again our traversal routine -- this time, to query the User in devel-  */
/* oping the Descriptor Fields:                                           */
hie-ptr = db-ptr->hdn-root-seg;
traverse-hierarchy(hie-ptr, f-ptr, RESTTIME);

/* Now, we will traverse the Linked List of Descriptor Fields and         */
/* Values which we've created, writing them to our Descriptor File:       */
descriptornode-ptr = desc-head-ptr;
while (descriptornode-ptr != NULL)
  {
  if (descriptornode-ptr->first-value-node != NULL)
    {
    fprintf(f-ptr->fi-fid, "%s %c\n", descriptornode-ptr->attr-name,
                          descriptornode-ptr->descriptor-type);
    valuenode-ptr = descriptornode-ptr->first-value-node;
    while (valuenode-ptr != NULL)
      {
      fprintf(f-ptr->fi-fid, "%s %s\n", valuenode-ptr->value1,
                            valuenode-ptr->value2);
      valuenode-ptr = valuenode-ptr->next-value-node;
      } /* End "while (valuenode-ptr != NULL)" */
    fprintf(f-ptr->fi-fid, "@\n");
    } /* End "if (descriptornode-ptr->first-value-node != NULL)" */
```

```
  descriptornode-ptr = descriptornode-ptr->next-desc-node:
  }  /* End "while (descriptornode-ptr != NULL)" */
fprintf(f-ptr->fi-fid. "$\n"):
}  /* End "build-desc-file()" routine */
```

```
traverse-hierarchy(hie-ptr, f-ptr, traversal-number)

  struct   hrec-node      *hie-ptr:        /* ptr to hierarchical node   *
  struct   file-info      *f-ptr:          /* file pointer               *
  int                   traversal-number; /* control information        */

  {
  /*  This routine performs a pre-order recursive traversal of an Hierarchi- */
  /*  cal data structure:                                                    */

  if (traversal-number == FIRSTTIME)
    print-segment-info(hie-ptr, f-ptr);
  else
    query-segment-info(hie-ptr);

  if (hie-ptr->hn-first-child != NULL)
    traverse-hierarchy(hie-ptr->hn-first-child, f-ptr, traversal-number);

  if (hie-ptr->hn-next-sib != NULL)
    traverse-hierarchy(hie-ptr->hn-next-sib, f-ptr, traversal-number);

  }  /*  End "traverse-hierarchy(...)" routine  */
```

```
print-segment-info(hie-ptr, f-ptr)

  struct   hrec-node      *hie-ptr;        /* ptr to hierarchical node  *
  struct   file-info      *f-ptr;          /* file pointer              *

  {
  int                  str-len,        /* length of current string  */
                       index;          /* loop index                */
  /*  This routine writes Segment Names as EQUALITY Descriptors to the      */
  /*  Descriptor File, while concurrently printing the names to the screen. */
  /*  Only the first character of the Segment Name should be in upper case; */
  /*  all other characters in the name must be lower case:                  */

  fprintf(f-ptr->fi-fid, "! ");
  fprintf(f-ptr->fi-fid, "%c", hie-ptr->hn-name[0]);
  str-len = strlen(hie-ptr->hn-name);
  for (index = 1; index < str-len; index++)
    if (isupper(hie-ptr->hn-name[index]))
      fprintf(f-ptr->fi-fid, "%c", tolower(hie-ptr->hn-name[index]));
    else
      fprintf(f-ptr->fi-fid, "%c", hie-ptr->hn-name[index]);
  fprintf(f-ptr->fi-fid, "\n");
  printf("\n\t%s", hie-ptr->hn-name);
  }  /*  End "print-segment-info(...)" routine  */
```

```
query-segment-info(hie-ptr)

  struct   hrec-node      *hie-ptr;        * ptr to hierarchical node  *

  {
  /*  This routine presents each Field of the current Segment to the      */
  /*  User, determining whether it is to be installed as a Descriptor At-  */
  /*  tribute -- and, if so, whether it is to be an EQUALITY or RANGE OF   */
  /*  VALUES Descriptor:                                                   */

  struct   hattr-node     *field-ptr;      /* ptr to Field node          */
  struct   value-node     *valuenode-ptr;  /* ptr to Value Node          */
  struct   descriptor-node *descriptornode-ptr, /* ptr to Descriptor List */
                  *descriptor-node-alloc(); /* Allocates Nodes          */
  int              found,          /* Boolean flag                      */
                   num,            /* user query response               */
                   goodanswer;     /* Boolean flag                      */

  /*  Begin by setting the Field pointer to the first Field of the        */
  /*  current Segment, then -- while there are more Fields to process --  */
  /*  display the current Segment and Field:                              */
  field-ptr = hie-ptr->hn-first-attr;
  while (field-ptr != NULL)
    {
    system("clear");
    printf("Segment name:  %s\n", hie-ptr->hn-name);
    printf("Field name:    %s\n",field-ptr->han-name);

    /*  Now, traverse the Field linked list that is being created, to     */
    /*  see if the current Field has already been established as a De-    */
    /*  scriptor Field.  If so, offer the User the opportunity to add     */
    /*  additional EQUALITY or RANGE OF VALUE values; otherwise, offer the */
    /*  User the opportunity to establish this as a Descriptor Field:     */
    descriptornode-ptr = desc-head-ptr;
    found = FALSE;
    while ((descriptornode-ptr != NULL) && (found == FALSE))
      {
      if (strcmp(field-ptr->han-name, descriptornode-ptr->attr-name) == 0)
        {

        /*  The Field HAS already been chosen as a Descriptor.  Allow    */
        /*  the User the option of adding additional Descriptor values,  */
        /*  after listing those already entered:                         */

        printf("\nThis Field has been chosen as an Indexing Field.\n");
        printf("The following are the values ");
        printf("that have been specified:\n\n");
        found = TRUE;
        valuenode-ptr = descriptornode-ptr->first-value-node;
        while (valuenode-ptr != NULL)
          {
```

```
        if (descriptornode-ptr->descriptor-type == 'A')
          printf("\t%s %s\n", valuenode-ptr->value1,
                        valuenode-ptr->value2);
        else
          printf("\t%s\n", valuenode-ptr->value2);
        valuenode-ptr = valuenode-ptr->next-value-node;
        }  /*  End "while (valuenode-ptr != NULL)"  */

      printf("\nDo you wish to add more ");
      if (descriptornode-ptr->descriptor-type == 'A')
        printf("RANGE");
      else
        printf("EQUALITY");
      printf(" values? (y or n)\n");
      dli-info-ptr->di-answer = get-ans(&num);
      if ((dli-info-ptr->di-answer == 'y') ||
         (dli-info-ptr->di-answer == 'Y'))

        /*  The User DOES wish to add more Descriptors to the        */
        /*  currently existing list:                                 */
        {
        if (descriptornode-ptr->descriptor-type == 'A')
          build-RAN-descrip(descriptornode-ptr, field-ptr->han-length);
        else
          build-EQ-descrip(descriptornode-ptr, field-ptr->han-length);
        }  /*  End "if ((dli-info-ptr->di-answer == 'y') ||
                    (dli-info-ptr->di-answer == 'Y'))"  */
      }  /*  End "if (strcmp(...) == 0)"  */
    descriptornode-ptr = descriptornode-ptr->next-desc-node;
    }  /*  End "while ((descriptornode-ptr != NULL) && (found...))"  */

  if (found == FALSE)

    /*  The Field has NOT previously been chosen as a Descriptor.      */
    /*  Allow the User the option of making this a Descriptor Field,   */
    /*  with appropriate Descriptor Values:                            */
    {
    printf("\nDo you wish to install this Field as an ");
    printf("Indexing Field?\n\n");
    printf("\t(n) - no:  continue with next Field/Segment\n");
    printf("\t(e) - yes; establish this as an EQUALITY Indexing Field\n");
    printf("\t(r) - yes; establish this an a RANGE Indexing Field\n");

    goodanswer = FALSE;
    while (goodanswer == FALSE)
      {
      dli-info-ptr->di-answer = get-ans(&num);

      switch(dli-info-ptr->di-answer)
        {
        case 'n':  /*  User does NOT want to use this as an Indexing  */
```

```
                /*  Field:                                  */
                goodanswer = TRUE:
                break:

        case 'e':   /*  User wants to use this an an EQUALITY Indexing */
                /*  Field:                                  */
                goodanswer = TRUE;
                descriptornode-ptr = descriptor-node-alloc();
                descriptornode-ptr->next-desc-node = desc-head-ptr;
                desc-head-ptr = descriptornode-ptr;
                strcpy(descriptornode-ptr->attr-name,
                     field-ptr->han-name);
                descriptornode-ptr->descriptor-type = 'B';
                descriptornode-ptr->first-value-node = NULL;
                build-EQ-descrip(descriptornode-ptr,
                             field-ptr->han-length);
                break;

        case 'r':   /*  User wants to use this as a RANGE Indexing    */
                /*  Field:                                  */
                goodanswer = TRUE;
                descriptornode-ptr = descriptor-node-alloc();
                descriptornode-ptr->next-desc-node = desc-head-ptr;
                desc-head-ptr = descriptornode-ptr;
                strcpy(descriptornode-ptr->attr-name,
                     field-ptr->han-name);
                descriptornode-ptr->descriptor-type = 'A';
                descriptornode-ptr->first-value-node = NULL;
                build-RAN-descrip(descriptornode-ptr,
                              field-ptr->han-length);
                break;

        default:    /*  User did not select a valid choice:          */
                printf("\nError - Invalid Operation Selected;\n");
                printf("Please pick again\n");
                break;
        }  /*  End Switch  */
      }  /*  End "while (goodanswer = FALSE)"  */
    }  /*  End "if (found = FALSE)"  */
  field-ptr = field-ptr->han-next-attr;
  }  /*  End "while (field-ptr != NULL)"  */
}  /*  End "query-segment-info(...)" routine  */
```

```
build-EQ-descrip(descriptornode-ptr. attr-length)

  struct    descriptor-node      *descriptornode-ptr; /* ptr to a desc.node*/
  int                            attr-length;        /* length of an attr */

  {
  /*  This routine builds the EQUALITY Descriptor list for the current      */
  /*  Field:                                                                */


  struct    value-node       *valuenode-ptr,     /*  Points to Value Node  */
                             *value-node-alloc(); /*  Allocates Value Nodes */
  int                        end-routine;        /*  Boolean flag          */
  int                        index;              /*  Loop Index            */
  int                        loop-count;         /*  Loop Index            */
  int                        val-count;          /*  Loop Index            */
  int                        str-len;            /*  Length of input       */
  char                       *temp-value;        /*  Holds answer          */
  char                       *var-str-alloc();   /*  Allocates Str.        */

  /*  Repetitively offer the user the opportunity to create EQUALITY de-    */
  /*  scriptors for the current Field, halting when the user enters an      */
  /*  empty carriage return ("<CR>").                                       */

  temp-value = var-str-alloc(attr-length + 1);
  end-routine = FALSE;
  while (end-routine == FALSE)
    {
    printf("\nEnter EQUALITY match value, or <CR> to exit:");
    readstr(stdin, temp-value);
    str-len = strlen( temp-value );
    for (index = 0; temp-value[index] == ' '; index++)
      ;
    if ( str-len != index )
      {
      valuenode-ptr = value-node-alloc(attr-length);
      valuenode-ptr->next-value-node = descriptornode-ptr->first-value-node;
      descriptornode-ptr->first-value-node = valuenode-ptr;
      strcpy(valuenode-ptr->value1, "!");

      /* Convert first character in temp-value to upper case if necessary: */
      if (islower(temp-value[index]))
        temp-value[index] = toupper( temp-value[index] );

      /* Convert remaining chars in temp-value to lower case if necessary: */
      for( loop-count = index + 1; loop-count <= str-len; loop-count++)
        if (isupper(temp-value[loop-count]))
          temp-value[loop-count] = tolower( temp-value[loop-count] );

      /* Store temp-value into value2 from index to end of string.         */
      val-count = 0;
```

```
    for( loop-count = index; loop-count <= str-len; loop-count--)
      valuenode-ptr->value2 val-count--] = temp-value[loop-count];
    valuenode-ptr->value2 val-count] = '\0';
    } /* End "if (str-len != index)" */
  else
    end-routine = TRUE;
  } /* End "while (end-routine == FALSE)" */
free(temp-value);
} /* End "build-EQ-descrip(...)" routine */
```

```
build-RAN-descrip(descriptornode-ptr. attr-length)

  struct    descriptor-node      *descriptornode-ptr:  * ptr to a desc.node*
  int                            attr-length;         * length of an attr */

  {
  /*  This routine builds the RANGE OF VALUEs Descriptor list for the      */
  /*  current Field:                                                       */


  struct   value-node       *valuenode-ptr,     /*  Points to Value Node  */
                           *value-node-alloc(); /*  Allocates Value Nodes */
  int                      end-routine;        /*  Boolean flag          */
  int                      good-upper-value;   /*  Boolean flag          */
  int                      index;              /*  Loop Index            */
  int                      loop-count;         /*  Loop Index            */
  int                      val-count;          /*  Loop Index            */
  int                      str-len;            /*  Length of Input       */
  char                     *temp-value;        /*  Holds Answer          */
  char                     *var-str-alloc();   /*  Allocates String      */

   *  Repetitively offer the user the opportunity to create RANGE OF VALUE  */
   *  Descriptors for the current Field, halting when the user enters       */
   *  an empty carriage return ("<CR>").                                    */

  temp-value = var-str-alloc(attr-length + 1);
  end-routine = FALSE;
  while (end-routine == FALSE)
    {
    printf("\nEnter Lower Bound, or <CR> to exit:");
    readstr(stdin, temp-value);
    str-len = strlen( temp-value );
    for (index = 0; temp-value[index] == ' '; index++)
      ;
    if ( str-len != index )
      {
      valuenode-ptr = value-node-alloc();
      valuenode-ptr->next-value-node = descriptornode-ptr->first-value-node;
      descriptornode-ptr->first-value-node = valuenode-ptr;

      /* Convert first character in temp-value to upper case if necessary: */
      if (islower(temp-value[index]))
        temp-value[index] = toupper( temp-value[index] );

      /* Convert remaining chars in temp-value to lower case if necessary: */
      for( loop-count = index + 1; loop-count <= str-len; loop-count++)
        if (isupper(temp-value[loop-count]))
          temp-value[loop-count] = tolower( temp-value[loop-count] );

      /* Store temp-value into value1 from index to end of string:        */
      val-count = 0;
```

```
    for( loop-count = index; loop-count <= str-len; loop-count--)
      valuenode-ptr->value1[val-count--] = temp-value[loop-count];
    valuenode-ptr->value1[val-count] = '\0';

    good-upper-value = FALSE;
    while (good-upper-value == FALSE)
      {
      printf("\nEnter Upper Bound:");
      readstr(stdin, temp-value);
      str-len = strlen(temp-value);
      for (index = 0; temp-value[index] == ' '; index++)
        ;
      if (str-len != index)
        {
        /* Convert first character in temp-value to upper case if nec: */
        if (islower(temp-value[index]))
          temp-value[index] = toupper( temp-value[index] );

        /* Convert remaining chars in temp-value to lower case if nec: */
        for( loop-count = index + 1; loop-count <= str-len; loop-count++)
          if (isupper(temp-value[loop-count]))
            temp-value[loop-count] = tolower( temp-value[loop-count] );

        /* Store temp-value into value1 from index to end of string:   */
        val-count = 0;
        for( loop-count = index; loop-count <= str-len; loop-count++)
          valuenode-ptr->value2[val-count++] = temp-value[loop-count];
        valuenode-ptr->value2[val-count] = '\0';

        good-upper-value = TRUE;
        }  /*  End "if (str-len != index)"  */
      else
        printf("\nYou must supply a non-blank Upper Bound.\n");
      }  /*  End "while (good-upper-value == FALSE)"  */
    }  /*  End "if (str-len != index)"  */
  else
    end-routine = TRUE;
  }  /*  End "while (end-routine == FALSE)"  */
}  /*  End "build-RAN-descrip(...)" routine  */
```

APPENDIX F - THE NETWORK DESCRIPTOR FILE

```
#include <stdio.h>
#include <strings.h>
#include <ctype.h>
#include "licommdata.def"
#include "dml.ext"
#include "lil.ext"

struct   descriptor-node  *desc-head-ptr;  /*  ptr to Descriptor head node  */

build-desc-file()

  {
  /*  This routine builds the Descriptor File to be used by the MBDS in the  */
  /*  creation of Indexing Clusters:                                          */

  struct   net-dbid-node   *db-ptr;        /* database pointer          */
  struct   nrec-node       *net-ptr;       /* ptr to Network Node       */
  struct   descriptor-node *descriptornode-ptr; /* ptr to descriptor node */
  struct   value-node      *valuenode-ptr;  /* pointer to Value Node     */
  struct   file-info       *f-ptr;         /* file pointer              */
  int                      num,            /* User query response       */
                           found,          /* Boolean flag              */
                           goodanswer,     /* Boolean flag              */
                           index;          /* loop index                */

  /*  Begin by setting the pointers to the dml-info data structure that is  */
  /*  maintained for each user of the system:                                */
  db-ptr = dml-info-ptr->dmi-curr-db.cdi-db.dn-net;
  f-ptr  = &(dml-info-ptr->dmi-ddl-files->ddli-desc);

  /*  Next, copy the filename where the MBDS Descriptor File information    */
  /*  will be stored.  This filename is a Constant, and was obtained from   */
  /*  licommdata.def:                                                        */
  strcpy(f-ptr->fi-fname, NDESCFname);

  /*  Now, open the Descriptor File to be created, for Write access:        */
  f-ptr->fi-fid = fopen(f-ptr->fi-fname, "w");

  /*  The next step is to traverse the Records of the Network File.  There  */
  /*  are two reasons for doing so:  First, to write the Record Names to the */
  /*  Descriptor File as EQUALITY Descriptors; this is done automatically   */
  /*  with any Network Database, forms a necessary part of the Descriptor   */
  /*  File created from such a Database, and requires no User involvement.  */
  /*  Second, it allows us to present the Record names (without their re-   */
  /*  spective Attributes) to the User, as a memory jog:                     */
  system("clear");
```

```
fprintf(f-ptr->fi-fid, "%s\n", db-ptr->ndn-name);
fprintf(f-ptr->fi-fid, "FILE B\n");
printf("\nThe following are the Records in the ");
printf("%s", db-ptr->ndn-name);
printf(" Database:\n\n");

/* Call a routine to traverse the Network Database, writing the Record   */
/* names to the File in the prescribed format, as well as printing them  */
/* to the screen. A Linked List is also initialized here, to be used     */
/* later to hold designated Descriptor Values:                           */
net-ptr = db-ptr->ndn-first-rec;
desc-head-ptr = NULL;
traverse-network(net-ptr, f-ptr, FIRSTTIME);

/* Each Descriptor Block must be followed by the "@" sign:               */
fprintf(f-ptr->fi-fid, "@\n");

/* Now, inform the user of the procedure that must be followed to create */
/* the Descriptor File:                                                  */
printf("\n\nBeginning with the first Record, we will present each");
printf("\nAttribute of that Record. You will be prompted as to whether");
printf("\nyou wish to include that Attribute as an Indexing Attribute,");
printf("\nand, if so, whether it is to be indexed based on strict");
printf("\nEQUALITY, or based on a RANGE OF VALUES.");
printf("\n\nStrike RETURN when ready to continue.");
dml-info-ptr->dmi-answer = get-ans(&num);

/* Re-set the Network Record pointer to the First Record, then call again */
/* our traversal routine -- this time, to query the User in developing   */
/* the Descriptor Attributes:                                            */
net-ptr = db-ptr->ndn-first-rec;
traverse-network(net-ptr, f-ptr, RESTTIME);

/* Now, we will traverse the Linked List of Descriptor Attributes and    */
/* Values which we've created, writing them to our Descriptor File:      */
descriptornode-ptr = desc-head-ptr;
while (descriptornode-ptr != NULL)
  {
  if (descriptornode-ptr->first-value-node != NULL)
    {
    fprintf(f-ptr->fi-fid, "%s %c\n", descriptornode-ptr->attr-name,
                          descriptornode-ptr->descriptor-type);
    valuenode-ptr = descriptornode-ptr->first-value-node;
    while (valuenode-ptr != NULL)
      {
      fprintf(f-ptr->fi-fid, "%s %s\n", valuenode-ptr->value1,
                            valuenode-ptr->value2);
      valuenode-ptr = valuenode-ptr->next-value-node;
      } /* End "while (valuenode-ptr != NULL)" */
    fprintf(f-ptr->fi-fid, "@\n");
    } /* End "if (descriptornode-ptr->first-value-node != NULL)" */
```

```
  descriptornode-ptr = descriptornode-ptr->next-desc-node;
  }  /*  End "while (descriptornode-ptr != NULL)"  *
fprintf(f-ptr->fi-fid, "$\n");
}  /*  End "build-desc-file()" routine  *
```

```
traverse-network(net-ptr, f-ptr, traversal-number)

  struct   nrec-node      *net-ptr;        /* ptr to Network Node      */
  struct   file-info      *f-ptr;          /* File Pointer             */
  int                     traversal-number; /* control information     */

  {
  /*  This routine traverses a Network data structure:                 */

  struct   nattr-node     *at-ptr;         /* ptr to Attribute Node    */

  while (net-ptr != NULL)
    {
    if (traversal-number == FIRSTTIME)
      print-record-info(net-ptr, f-ptr);
    else
      {
      at-ptr = net-ptr->nrn-first-attr;
      query-record-info(at-ptr, net-ptr);
      }
    net-ptr = net-ptr->nrn-next-rec;
    } /*  End "while (net-ptr != NULL)"  */

  } /*  End "traverse-network(...)" routine  */
```

```
print-record-info(net-ptr, f-ptr)

  struct   nrec-node      *net-ptr;       /*  ptr to Network Node     *
  struct   file-info      *f-ptr:         /*  File Pointer            *

  {
  int                  str-len,       /* length of current string  */
                       index;         /* loop index                */

  /*  This routine writes Record names as EQUALITY Descriptors to the      */
  /*  Descriptor File, while concurrently printing the names to the screen: */

  fprintf(f-ptr->fi-fid, "! ");
  fprintf(f-ptr->fi-fid, "%c", net-ptr->nrn-name[0]);
  str-len = strlen(net-ptr->nrn-name);
  for (index = 1; index < str-len; index++)
    if (isupper(net-ptr->nrn-name[index]))
      fprintf(f-ptr->fi-fid, "%c", tolower(net-ptr->nrn-name[index]));
    else
      fprintf(f-ptr->fi-fid, "%c", net-ptr->nrn-name[index]);
  fprintf(f-ptr->fi-fid, "\n");
  printf("\n\t%s", net-ptr->nrn-name);
  }  /*  End "print-record-info(...)" routine  */
```

```
query-record-info(at-ptr. net-ptr)

  struct   nattr-node      *at-ptr;        /* ptr to attribute node     */
  struct   nrec-node       *net-ptr;       /* ptr to record node        */

  {
  /*  This routine performs a pre-order traversal of the Network Attribute  */
  /*  data structure:                                                       */

  get-record-info(at-ptr, net-ptr);

  if (at-ptr->nan-child != NULL)
    query-record-info(at-ptr->nan-child, net-ptr);
  if (at-ptr->nan-next-attr != NULL)
    query-record-info(at-ptr->nan-next-attr, net-ptr);
  }  /*  End "query-record-info(...)" routine  */
```

```
get-record-info(at-ptr. net-ptr)

  struct   nattr-node     *at-ptr;        /* ptr to attribute node     *
  struct   nrec-node      *net-ptr:       /* ptr to record node        *

  {
  /*  This routine presents each Attribute of the current Record to the     */
  /*  User, determining whether it is to be installed as a Descriptor At-   */
  /*  tribute -- and, if so. whether it is to be an EQUALITY or RANGE OF    */
  /*  VALUES Descriptor:                                                    */

  struct   value-node     *valuenode-ptr;     /* ptr to Value Node     */
  struct   descriptor-node *descriptornode-ptr, /* ptr to Descriptor List */
                   *descriptor-node-alloc(); /* Allocates Nodes    */
  int               found,          /* Boolean flag          */
                    num,            /* user query response    */
                    goodanswer:     /* Boolean flag          */

  /*  Begin by printing the current Record and Attribute Names on the       */
  /*  screen:                                                               */
  system("clear");
  printf("Record name:    %s\n", net-ptr->nrn-name);
  printf("Attribute name:  %s\n",at-ptr->nan-name);

  /*  Now, traverse the Attribute linked list that is being created, to     */
  /*  see if the current Attribute has already been established as a De-    */
  /*  scriptor Attribute.  If so, offer the User the opportunity to add     */
  /*  additional EQUALITY or RANGE OF VALUE values: otherwise, offer the    */
  /*  User the opportunity to establish this as a Descriptor Attribute:     */
  descriptornode-ptr = desc-head-ptr;
  found = FALSE;
  while ((descriptornode-ptr != NULL) && (found == FALSE))
    {
    if (strcmp(at-ptr->nan-name, descriptornode-ptr->attr-name) == 0)
      {

      /*  The Attribute HAS already been chosen as a Descriptor.  Allow     */
      /*  the User the option of adding additional Descriptor values,       */
      /*  after listing those already entered:                              */

      printf("\nThis Attribute has been chosen as an Indexing Attribute.\n");
      printf("The following are the values that have been specified:\n\n");
      found = TRUE;
      valuenode-ptr = descriptornode-ptr->first-value-node;
      while (valuenode-ptr != NULL)
        {
        if (descriptornode-ptr->descriptor-type == 'A')
          printf("\t%s %s\n", valuenode-ptr->value1,
                          valuenode-ptr->value2);
        else
          printf("\t%s\n", valuenode-ptr->value2);
```

```
      valuenode-ptr = valuenode-ptr->next-value-node;
      } /* End "while (valuenode-ptr != NULL)" */

    printf("\nDo you wish to add more ");
    if (descriptornode-ptr->descriptor-type == 'A')
      printf("RANGE");
    else
      printf("EQUALITY");
    printf(" values? (y or n)\n");
    dml-info-ptr->dmi-answer = get-ans(&num);
    if ((dml-info-ptr->dmi-answer == 'y') ||
       (dml-info-ptr->dmi-answer == 'Y'))

      /* The User DOES wish to add more Descriptors to the          */
      /* currently existing list:                                   */
      {
      if (descriptornode-ptr->descriptor-type == 'A')
        build-RAN-descrip(descriptornode-ptr, at-ptr->nan-length1);
      else
        build-EQ-descrip(descriptornode-ptr, at-ptr->nan-length1);
      } /* End "if ((dml-info-ptr->dmi-answer == 'y') ||
                  (dml-info-ptr->dmi-answer == 'Y'))" */
    } /* End "if (strcmp(...) == 0)" */
  descriptornode-ptr = descriptornode-ptr->next-desc-node;
  } /* End "while ((descriptornode-ptr != NULL) && (found...))" */

if (found == FALSE)

  /* The Attribute has NOT previously been chosen as a Descriptor.    */
  /* Allow the User the option of making this a Descriptor Attribute, */
  /* with appropriate Descriptor Values:                              */
  {
  printf("\nDo you wish to install this Attribute as an ");
  printf("Indexing Attribute?\n\n");
  printf("\t(n) - no: continue with next Attribute/Record\n");
  printf("\t(e) - yes; establish this as an EQUALITY Indexing Attribute\n");
  printf("\t(r) - yes; establish this as a RANGE Indexing Attribute\n");

  goodanswer = FALSE;
  while (goodanswer == FALSE)
    {
    dml-info-ptr->dmi-answer = get-ans(&num);

    switch(dml-info-ptr->dmi-answer)
      {
      case 'n':  /* User does NOT want to use this as an Indexing    */
                 /* Attribute:                                       */
                 goodanswer = TRUE;
                 break;

      case 'e':  /* User wants to use this an an EQUALITY Indexing   */
```

```
                /*  Attribute:                                  */
                goodanswer = TRUE;
                descriptornode-ptr = descriptor-node-alloc();
                descriptornode-ptr->next-desc-node = desc-head-ptr;
                desc-head-ptr = descriptornode-ptr;
                strcpy(descriptornode-ptr->attr-name,
                     at-ptr->nan-name);
                descriptornode-ptr->descriptor-type = 'B';
                descriptornode-ptr->first-value-node = NULL;
                build-EQ-descrip(descriptornode-ptr,
                          at-ptr->nan-length1);
                break;

        case 'r':  /*  User wants to use this as a RANGE Indexing      */
                /*  Attribute:                                  */
                goodanswer = TRUE;
                descriptornode-ptr = descriptor-node-alloc();
                descriptornode-ptr->next-desc-node = desc-head-ptr;
                desc-head-ptr = descriptornode-ptr;
                strcpy(descriptornode-ptr->attr-name,
                     at-ptr->nan-name);
                descriptornode-ptr->descriptor-type = 'A';
                descriptornode-ptr->first-value-node = NULL;
                build-RAN-descrip(descriptornode-ptr,
                          at-ptr->nan-length1);
                break:

        default:   /*  User did not select a valid choice:          */
                printf("\nError - Invalid Operation Selected;\n");
                printf("Please pick again\n");
                break;
        }  /*  End Switch  */
      }  /*  End "while (goodanswer = FALSE)"  */
    }  /*  End "if (found = FALSE)"  */

}  /*  End "get-record-info(...)" routine  */
```

```
build-EQ-descrip(descriptornode-ptr, attr-length)

  struct    descriptor-node     *descriptornode-ptr;  * ptr to a desc.node*
  int                           attr-length;         * length of an attr *

  {
  /*  This routine builds the EQUALITY Descriptor list for the current      */
  /*  Field:                                                                */


  struct   value-node      *valuenode-ptr,     /*  Points to Value Node  */
                          *value-node-alloc(); /*  Allocates Value Nodes */
  int                      end-routine;        /*  Boolean flag          */
  int                      index;              /*  Loop Index            */
  int                      loop-count;         /*  Loop Index            */
  int                      val-count;          /*  Loop Index            */
  int                      str-len;            /*  Length of input       */
  char                     *temp-value;        /*  Holds answer          */
  char                     *var-str-alloc();   /*  Allocates Str.        */

  /*  Repetitively offer the user the opportunity to create EQUALITY de-    */
  /*  scriptors for the current Field, halting when the user enters an      */
  /*  empty carriage return ("<CR>").                                       */

  temp-value = var-str-alloc(attr-length + 1);
  end-routine = FALSE;
  while (end-routine == FALSE)
    {
    printf("\nEnter EQUALITY match value, or <CR> to exit:");
    readstr(stdin, temp-value);
    str-len = strlen( temp-value );
    for (index = 0; temp-value[index] == ' '; index++)
      ;
    if ( str-len != index )
      {
      valuenode-ptr = value-node-alloc(attr-length);
      valuenode-ptr->next-value-node = descriptornode-ptr->first-value-node;
      descriptornode-ptr->first-value-node = valuenode-ptr;
      strcpy(valuenode-ptr->value1, "!");

      /* Convert first character in temp-value to upper case if necessary: */
      if (islower(temp-value[index]))
        temp-value[index] = toupper( temp-value[index] );

      /* Convert remaining chars in temp-value to lower case if necessary: */
      for( loop-count = index + 1; loop-count <= str-len; loop-count++)
       if (isupper(temp-value[loop-count]))
         temp-value[loop-count] = tolower( temp-value[loop-count] );

      /* Store temp-value into value2 from index to end of string.         */
      val-count = 0;
```

```
    for( loop-count = index: loop-count <= str-len: loop-count--)
      valuenode-ptr->value2 val-count--  = temp-value loop-count :
    valuenode-ptr->value2 val-count  = '0':
    }  /*  End "if (str-len != index)"  */
  else
    end-routine = TRUE:
  }  /*  End "while (end-routine == FALSE)"  */
free(temp-value);
}  /*  End "build-EQ-descrip(...)" routine  */
```

```
build-RAN-descrip(descriptornode-ptr. attr-length)

  struct    descriptor-node    *descriptornode-ptr; /* ptr to a desc.node*
  int                          attr-length;        /* length of an attr *

  {
  /*  This routine builds the RANGE OF VALUEs Descriptor list for the      */
  /*  current Field:                                                       */


  struct    value-node       *valuenode-ptr,     /*  Points to Value Node  */
                             *value-node-alloc(); /*  Allocates Value Nodes */
  int                        end-routine;        /*  Boolean flag          */
  int                        good-upper-value;   /*  Boolean flag          */
  int                        index;              /*  Loop Index            */
  int                        loop-count;         /*  Loop Index            */
  int                        val-count;          /*  Loop Index            */
  int                        str-len;            /*  Length of Input       */
  char                       *temp-value;        /*  Holds Answer          */
  char                       *var-str-alloc();   /*  Allocates String      */

  /*  Repetitively offer the user the opportunity to create RANGE OF VALUE  */
  /*  Descriptors for the current Field, halting when the user enters       */
  /*  an empty carriage return ("<CR>").                                    */

  temp-value = var-str-alloc(attr-length + 1);
  end-routine = FALSE;
  while (end-routine == FALSE)
    {
    printf("\nEnter Lower Bound, or <CR> to exit:");
    readstr(stdin, temp-value);
    str-len = strlen( temp-value );
    for (index = 0; temp-value[index] == ' '; index++)
      ;
    if ( str-len != index )
      {
      valuenode-ptr = value-node-alloc();
      valuenode-ptr->next-value-node = descriptornode-ptr->first-value-node;
      descriptornode-ptr->first-value-node = valuenode-ptr;

      /* Convert first character in temp-value to upper case if necessary: */
      if (islower(temp-value[index]))
        temp-value[index] = toupper( temp-value[index] );

      /* Convert remaining chars in temp-value to lower case if necessary: */
      for( loop-count = index + 1; loop-count <= str-len; loop-count++)
        if (isupper(temp-value[loop-count]))
          temp-value[loop-count] = tolower( temp-value[loop-count] );

      /* Store temp-value into value1 from index to end of string:        */
      val-count = 0;
```

```
    for( loop-count = index; loop-count <= str-len; loop-count++)
      valuenode-ptr->value1[val-count++] = temp-value[loop-count];
    valuenode-ptr->value1[val-count] = '\0';

    good-upper-value = FALSE;
    while (good-upper-value == FALSE)
      {
      printf("\nEnter Upper Bound:");
      readstr(stdin, temp-value);
      str-len = strlen(temp-value);
      for (index = 0; temp-value[index] == ' '; index++)
        ;
      if (str-len != index)
        {
        /* Convert first character in temp-value to upper case if nec: */
        if (islower(temp-value[index]))
          temp-value[index] = toupper( temp-value[index] );

        /* Convert remaining chars in temp-value to lower case if nec: */
        for( loop-count = index + 1; loop-count <= str-len; loop-count++)
          if (isupper(temp-value[loop-count]))
            temp-value[loop-count] = tolower( temp-value[loop-count] );

        /* Store temp-value into value1 from index to end of string:  */
        val-count = 0;
        for( loop-count = index; loop-count <= str-len; loop-count++)
          valuenode-ptr->value2[val-count++] = temp-value[loop-count];
        valuenode-ptr->value2[val-count] = '\0';

        good-upper-value = TRUE;
        } /*  End "if (str-len != index)"  */
      else
        printf("\nYou must supply a non-blank Upper Bound.\n");
      } /*  End "while (good-upper-value == FALSE)"  */
    } /*  End "if (str-len != index)"  */
  else
    end-routine = TRUE;
  } /*  End "while (end-routine == FALSE)"  */
} /*  End "build-RAN-descrip(...)" routine  */
```

LIST OF REFERENCES

1. Hsiao, D. K., and Harary, F., "A Formal System for Information Retrieval from Files." *Communications of the ACM.* Vol. 13, No. 2, February 1970, also in Corrigenda, Vol. 13, No. 3, March 1970.

2. Wong, E., and Chiang, T. C., "Canonical Structure in Attribute Based File Organization," *Communications of the ACM.* Vol. 14, No. 9, September 1971.

3. Naval Postgraduate School, Monterey, California, Technical Report, NPS52-86-011, *The Multi-Lingual Database System*, by S. A. Demurjian and D. K. Hsiao, February 1986.

4. Naval Postgraduate School, Monterey, California, Technical Report, NPS52-85-009, *Design, Analysis and Performance Evaluation Methodologies for Database Computers*, by S. A. Demurjian, D. K. Hsiao and P. R. Strawser, June 1985.

5. The Ohio State University, Columbus, Ohio, Technical Report, OSU-CISRC-TR-81-7, *Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part I)*, by D. K. Hsiao and M. J. Menon, July 1981.

6. The Ohio State University, Columbus, Ohio, Technical Report, OSU-CISRC-TR-81-8, *Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part II)*, by D. K. Hsiao and M. J. Menon, July 1981.

7. Naval Postgraduate School, Monterey, California, Technical Report, NPS52-85-002, *A Multi-Backend Database System for Performance Gains, Capacity Growth and Hardware Gains*, by S. A. Demurjian, D. K. Hsiao and M. J. Menon, February 1985.

8. Wortherly, C. R., *The Design and Analysis of a Network Interface for the Multi-Lingual Database System*, M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

9. Kloepping, G. R. and Mack, J. F., *The Design and Implementation of a Relational Interface for the Multi-Lingual Database System*, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

10. Benson, T. P. and Wentz, G. L., *The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System*, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

11. Brooks. Jr.. Frederick P.. *the mythical man-month*. p. 16. Addison-Wesley Publishing Company. 1975.

12. Banerjee. J.. Baum. R. I.. and Hsiao. D. K.. "Concepts and capabilities of a database computer." *ACM Transactions on Database Systems*. Vol. 3. No. 4. December 1978.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5000	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
4.	Curricular Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5000	1
5.	Professor David K. Hsiao, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
6.	Mr. Steven A. Demurjian, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
7.	Captain Steven T. Holste 4160 - 128th Ave. S.E. #A-110 Bellevue, Washington 98006	3